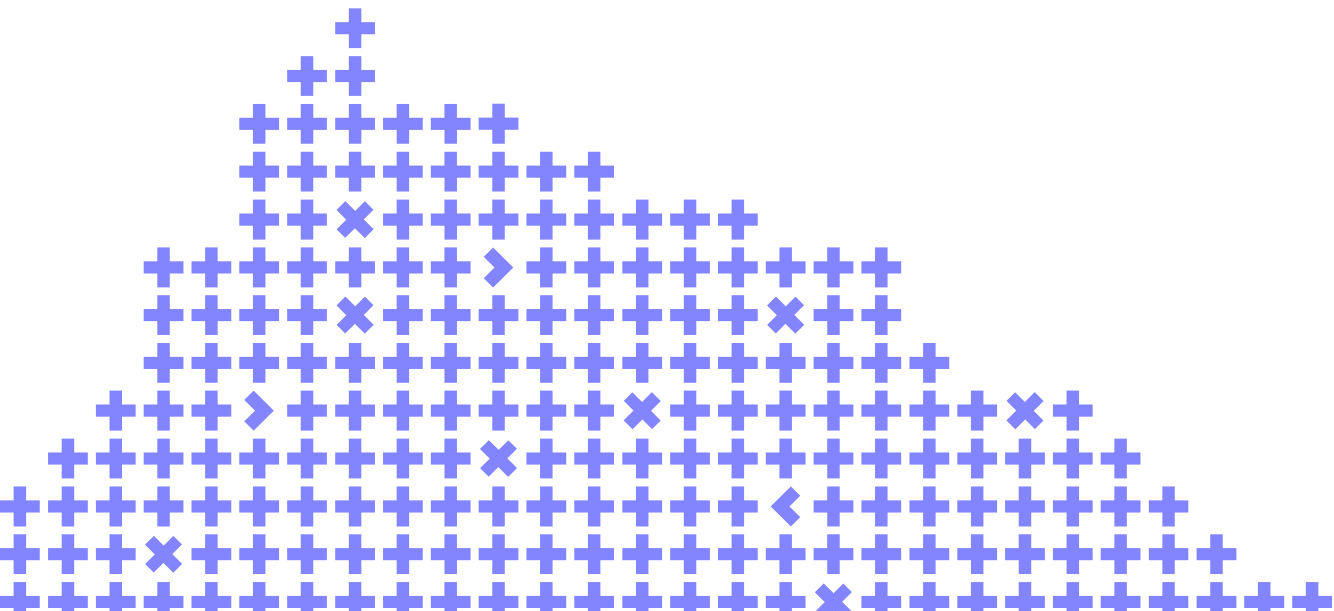


Fair threaded task scheduler verified in TLA+

Vladislav Shpilevoy



Co-organizer

Yandex

Plan

Task scheduling

Typical solutions

New task scheduler

Scheduling gears

Verification

Benchmarks

Future plans

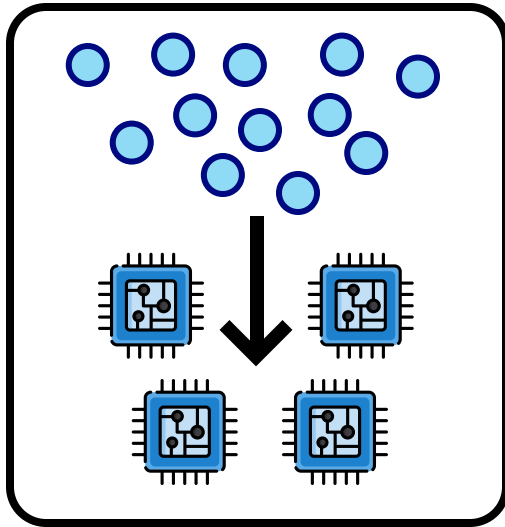
Task scheduling

is just execution of code

Task scheduling

is just execution of code

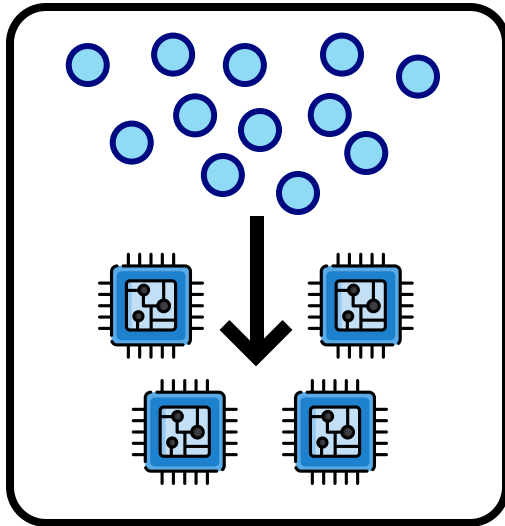
Thread pool



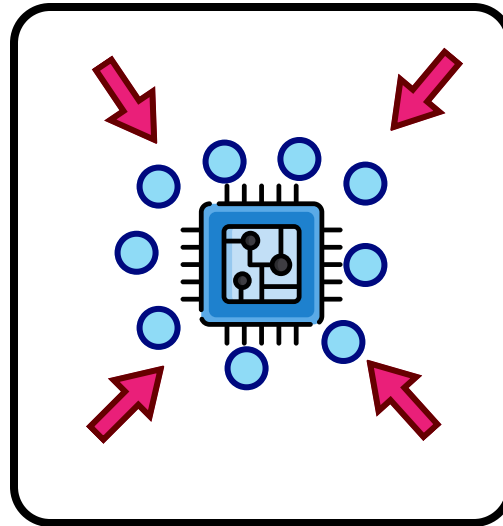
Task scheduling

is just execution of code

Thread pool



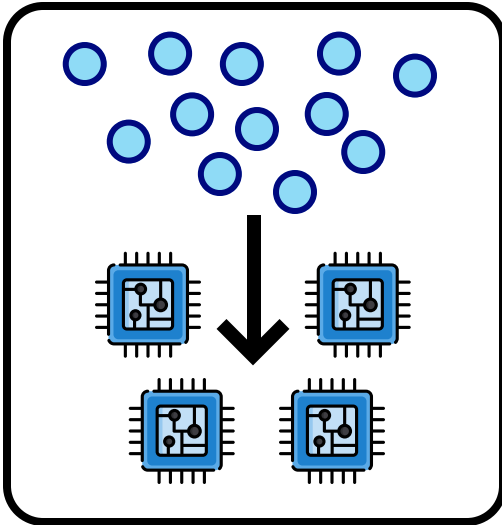
Event loop



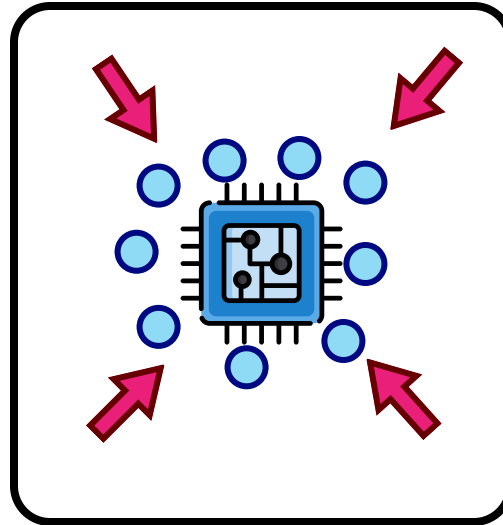
Task scheduling

is just execution of code

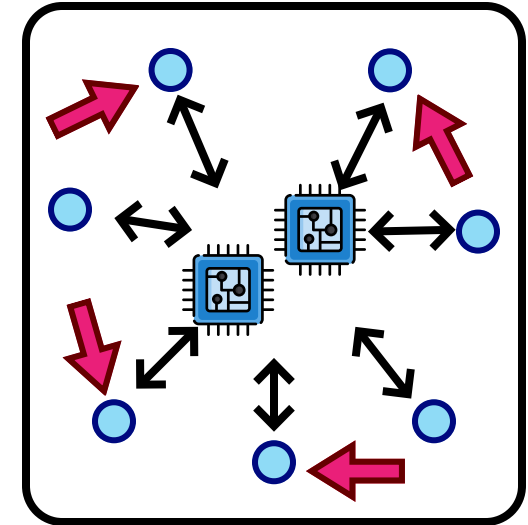
Thread pool



Event loop



Coroutine engine



Trivial scheduler

```
class TrivialSched:
    Thread          myThread;
    Mutex           myLock;
    ConditionVariable myCond;
    List<Callback>  myQueue;
```


```
TrivialSched::Post(const Callback& aFunc)
{
    myLock.Lock();
    myQueue.Append(aFunc);
    myCond.Signal();
    myLock.Unlock();
}

TrivialSched::Run()
{
    while(!IsStopped()) {
        myLock.Lock();
        while (myQueue.IsEmpty())
            myCond.Wait();
        Callback cb = myQueue.Pop();
        myLock.Unlock();

        cb.Execute();
    }
}
```


Trivial scheduler

```
class TrivialSched:  
    Thread          myThread;  
    Mutex          myLock;  
    ConditionVariable myCond;  
    List<Callback> myQueue;
```



Single thread

```
TrivialSched::Post(const Callback& aFunc)  
{  
    myLock.Lock();  
    myQueue.Append(aFunc);  
    myCond.Signal();  
    myLock.Unlock();  
}  
  
TrivialSched::Run()  
{  
    while(!IsStopped()) {  
        myLock.Lock();  
        while (myQueue.IsEmpty())  
            myCond.Wait();  
        Callback cb = myQueue.Pop();  
        myLock.Unlock();  
  
        cb.Execute();  
    }  
}
```



Trivial scheduler

```
class TrivialSched:
    Thread          myThread;
    Mutex           myLock;
    ConditionVariable myCond;
    List<Callback>  myQueue;
```

Single thread

Simple locked
queue

```
TrivialSched::Post(const Callback& aFunc)
```

```
{
    myLock.Lock();
    myQueue.Append(aFunc);
    myCond.Signal();
    myLock.Unlock();
}
```

Append
under a lock

```
TrivialSched::Run()
```

```
{
    while(!IsStopped()) {
        myLock.Lock();
        while (myQueue.IsEmpty())
            myCond.Wait();
        Callback cb = myQueue.Pop();
        myLock.Unlock();

        cb.Execute();
    }
}
```

Pop under a
lock

Trivial scheduler

```
class TrivialSched:
    Thread          myThread;
    Mutex           myLock;
    ConditionVariable myCond;
    List<Callback>  myQueue;
```

Single thread

Simple locked
queue

Grind the tasks
one by one

```
TrivialSched::Post(const Callback& aFunc)
{
    myLock.Lock();
    myQueue.Append(aFunc);
    myCond.Signal();
    myLock.Unlock();
}

TrivialSched::Run()
{
    while(!IsStopped()) {
        myLock.Lock();
        while (myQueue.IsEmpty())
            myCond.Wait();
        Callback cb = myQueue.Pop();
        myLock.Unlock();

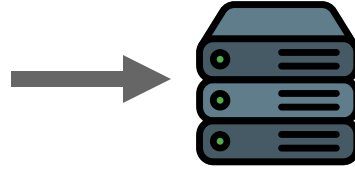
        cb.Execute();
    }
}
```

← Execute one by one

The real task

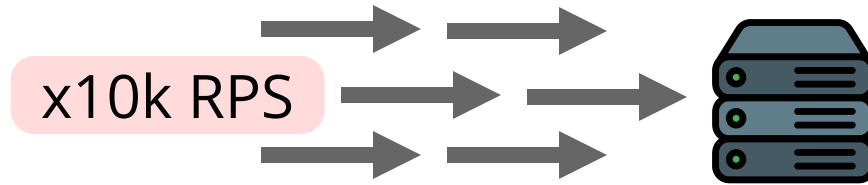
The real task

Handling savegames



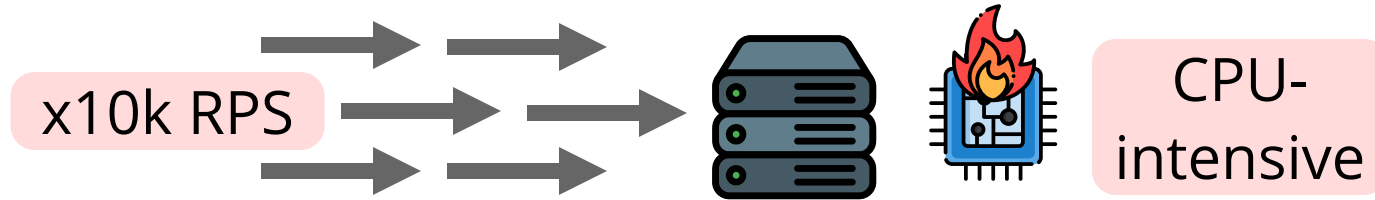
The real task

Handling savegames

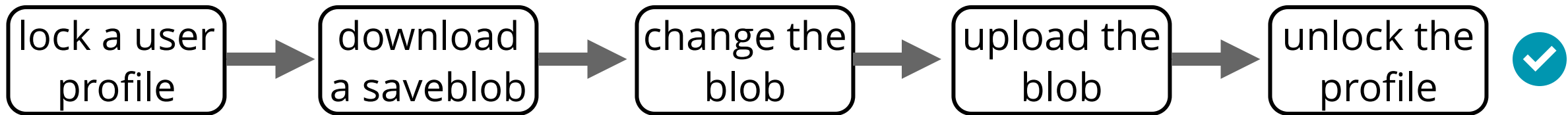


The real task

Handling savegames

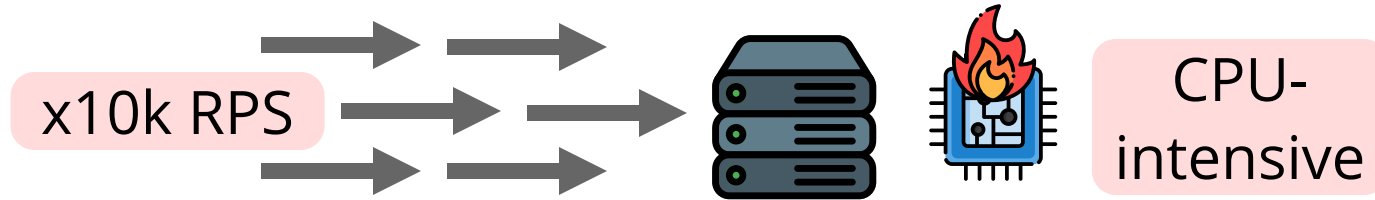


Multiple steps:

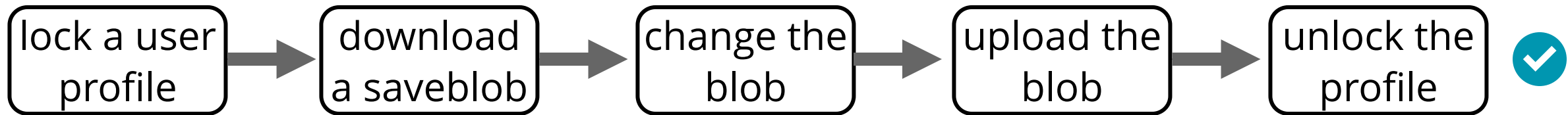


The real task

Handling savegames



Multiple steps:



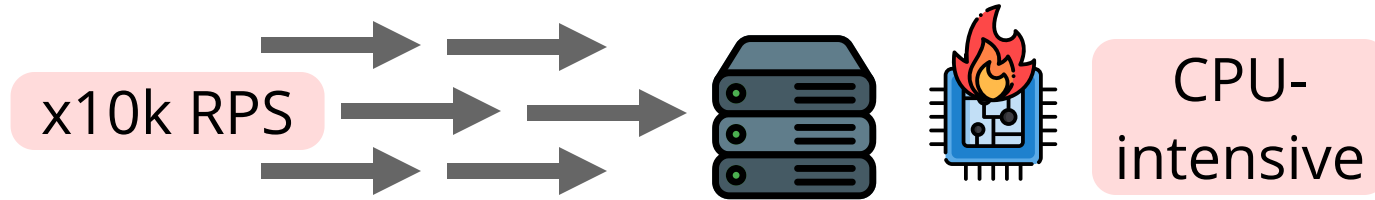
Network
exchange



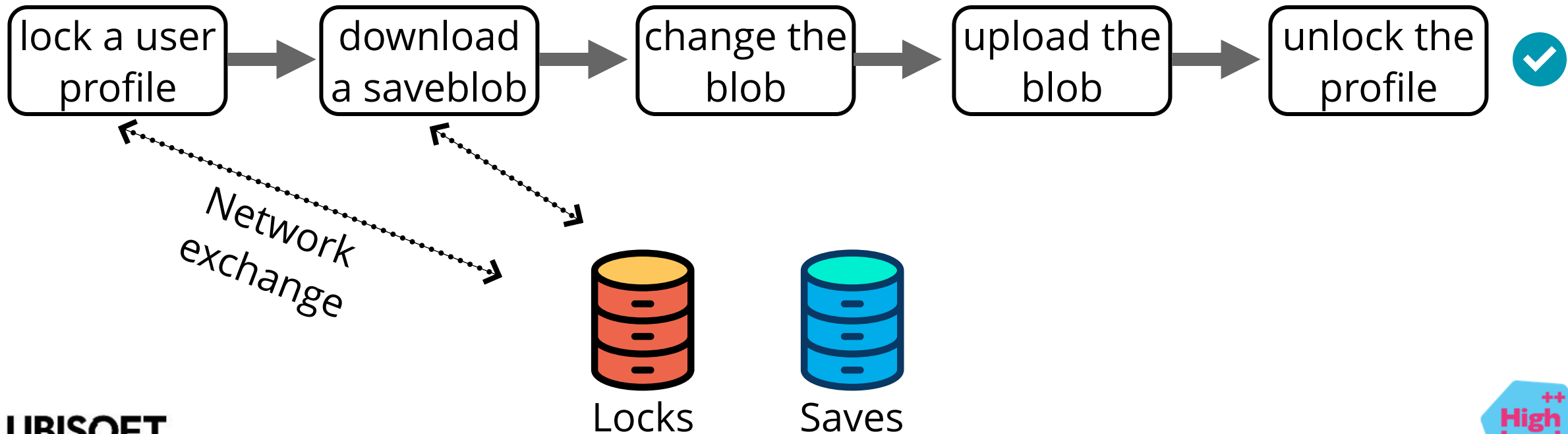
Locks

The real task

Handling savegames

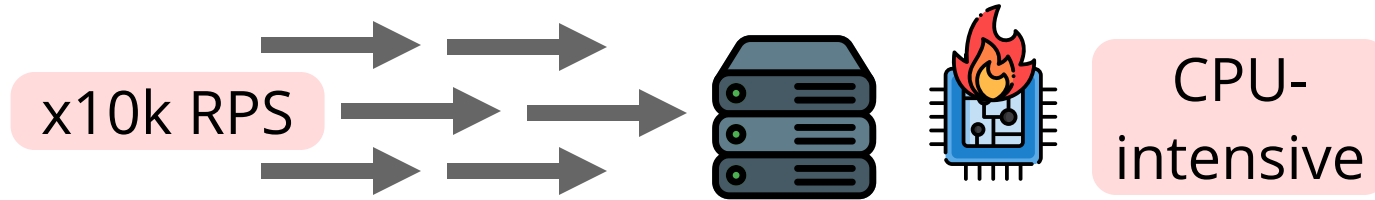


Multiple steps:

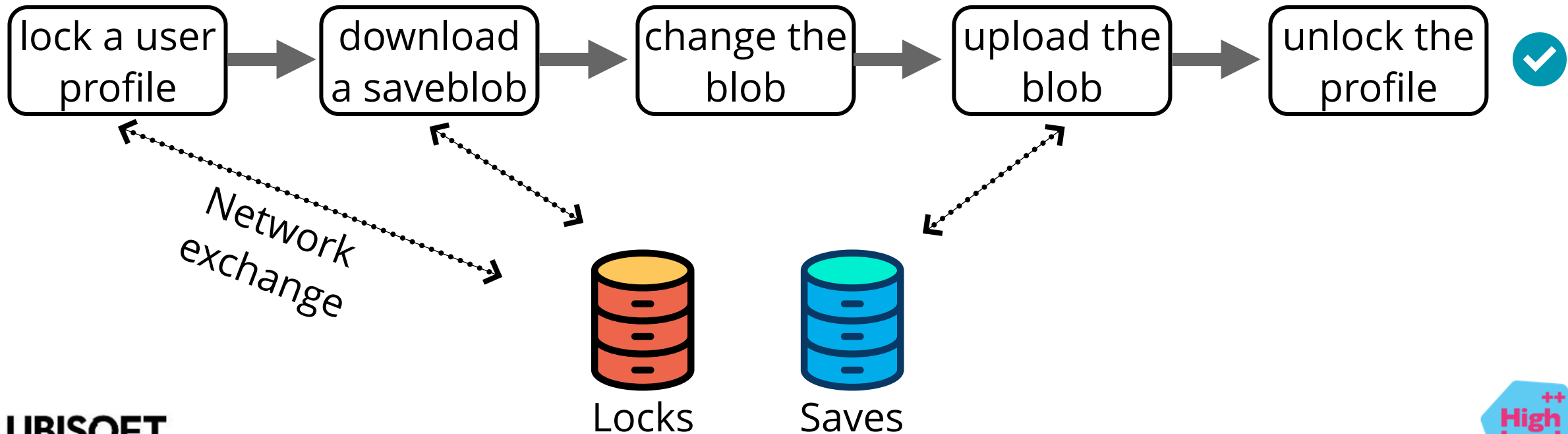


The real task

Handling savegames

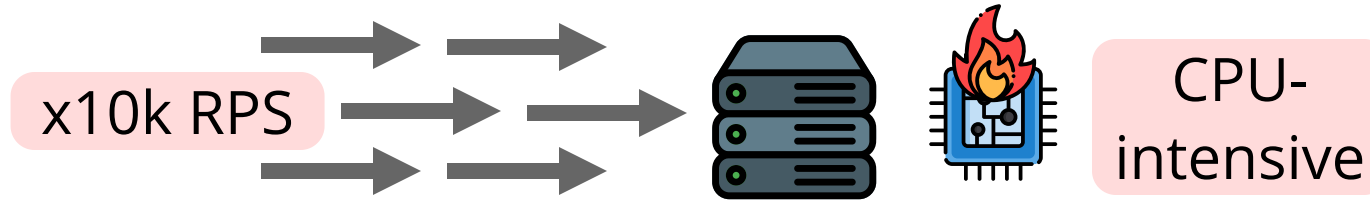


Multiple steps:

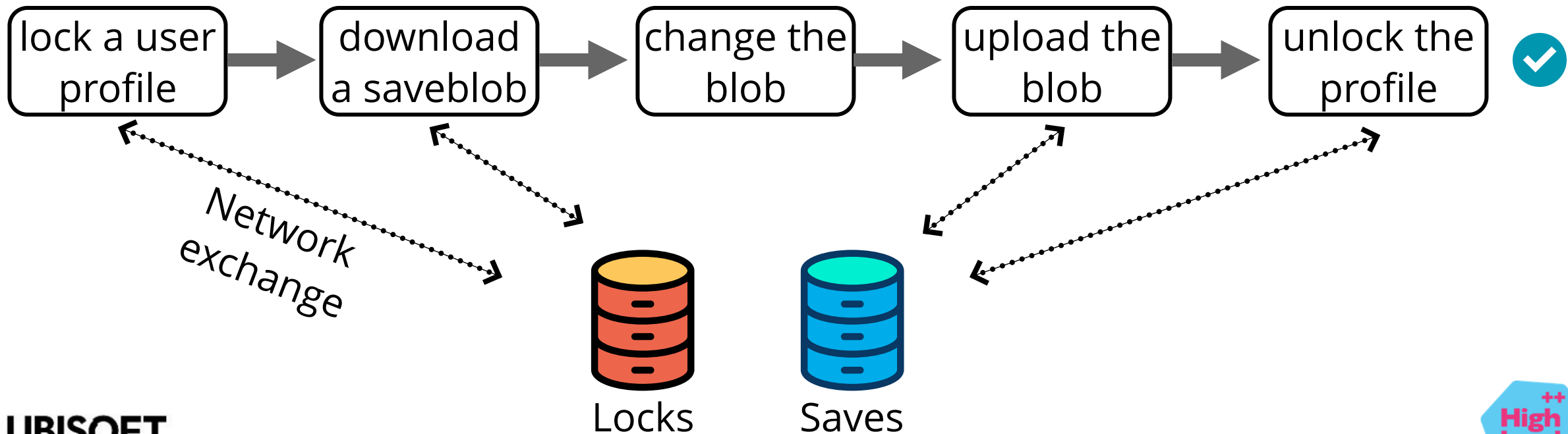


The real task

Handling savegames



Multiple steps:



Trivial scheduler problems

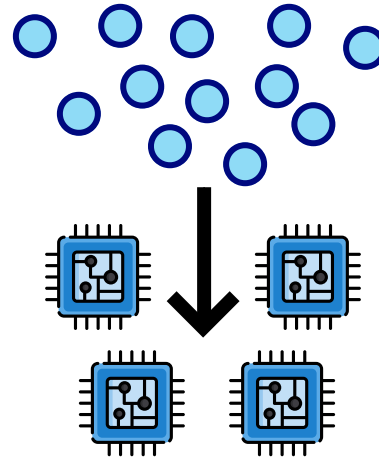
Won't scale

Trivial scheduler problems

Won't scale

Need multiple threads

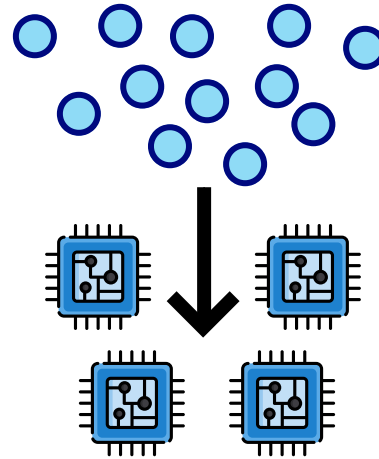
Choking on CPU



Trivial scheduler problems

Won't scale

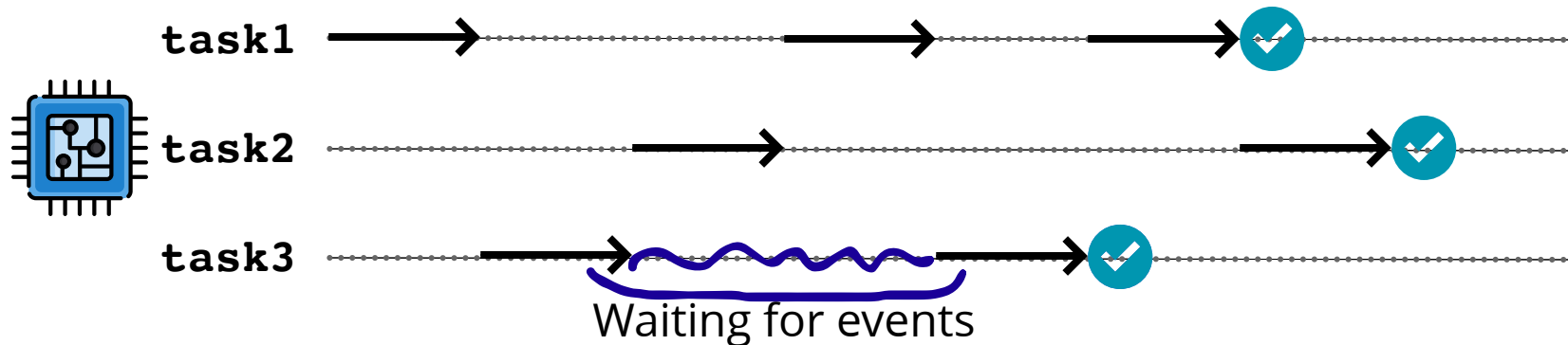
Need multiple threads



Choking on CPU

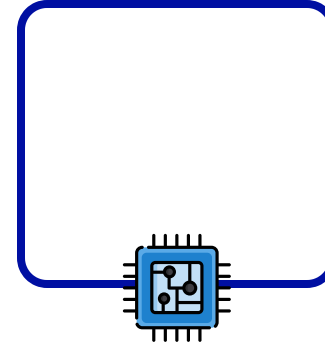
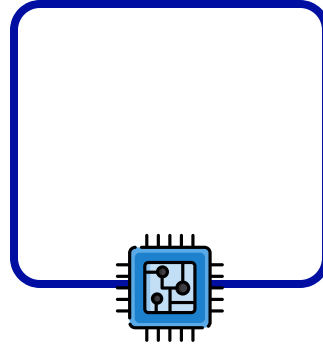
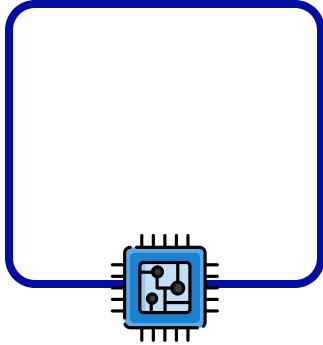
Can't postpone a blocked task

Need coroutines



Trivial scheduler v2

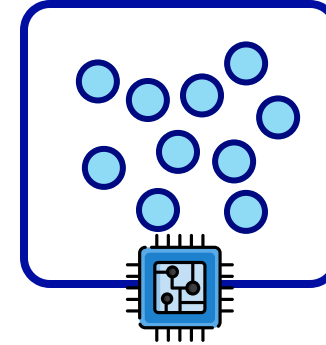
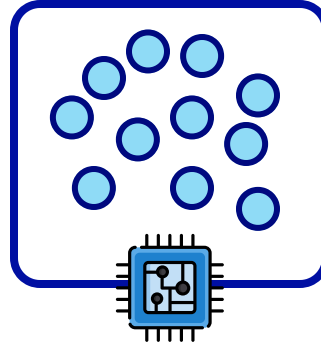
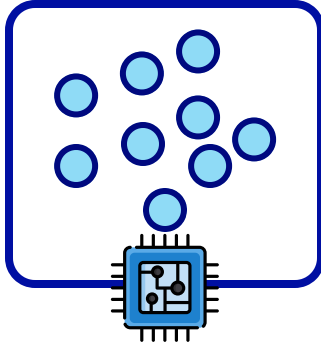
Trivial scheduler v2



Thread pool

Trivial scheduler v2

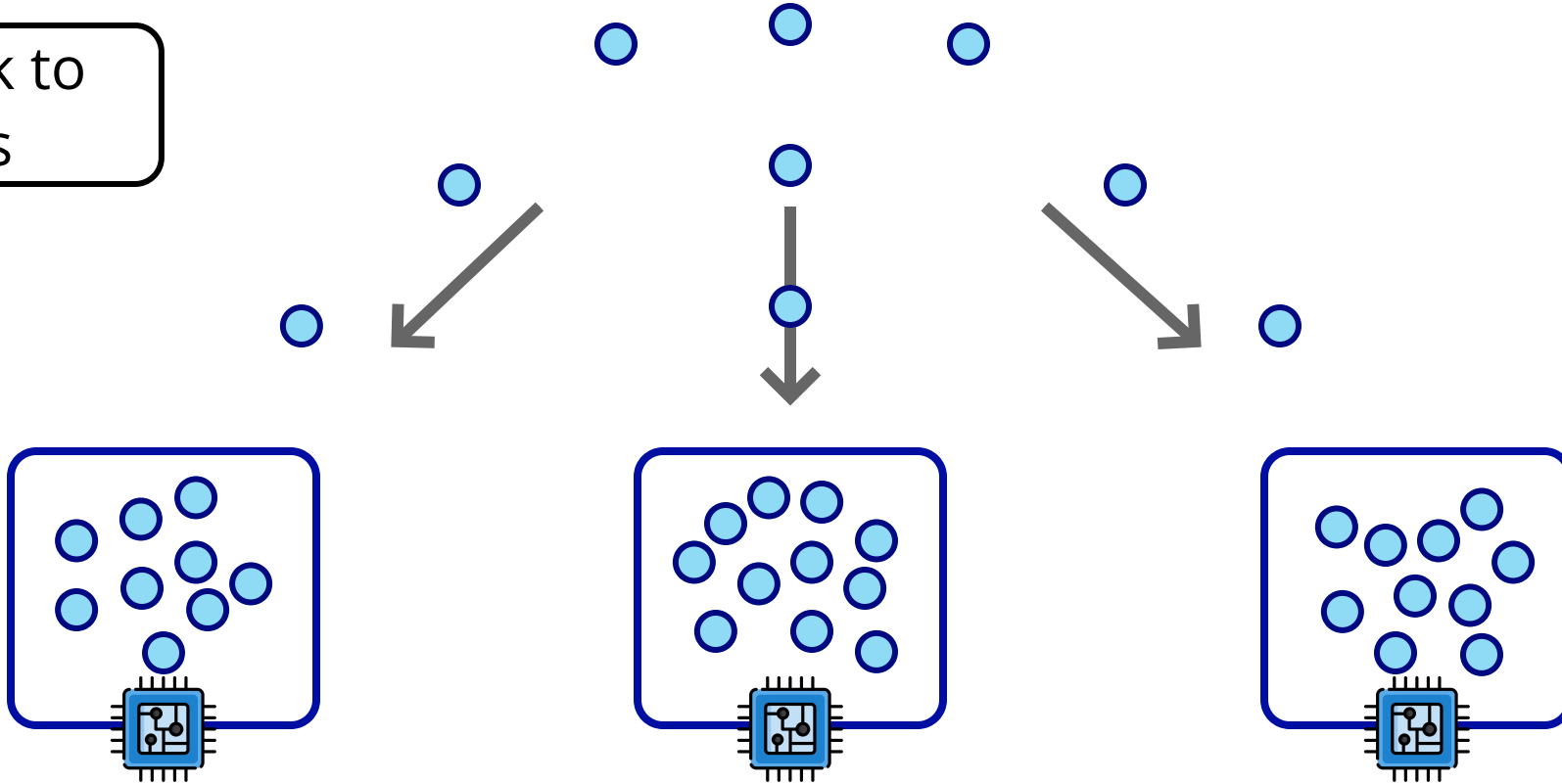
Tasks stick to
threads



Trivial scheduler v2

Round-robin distribution

Tasks stick to
threads

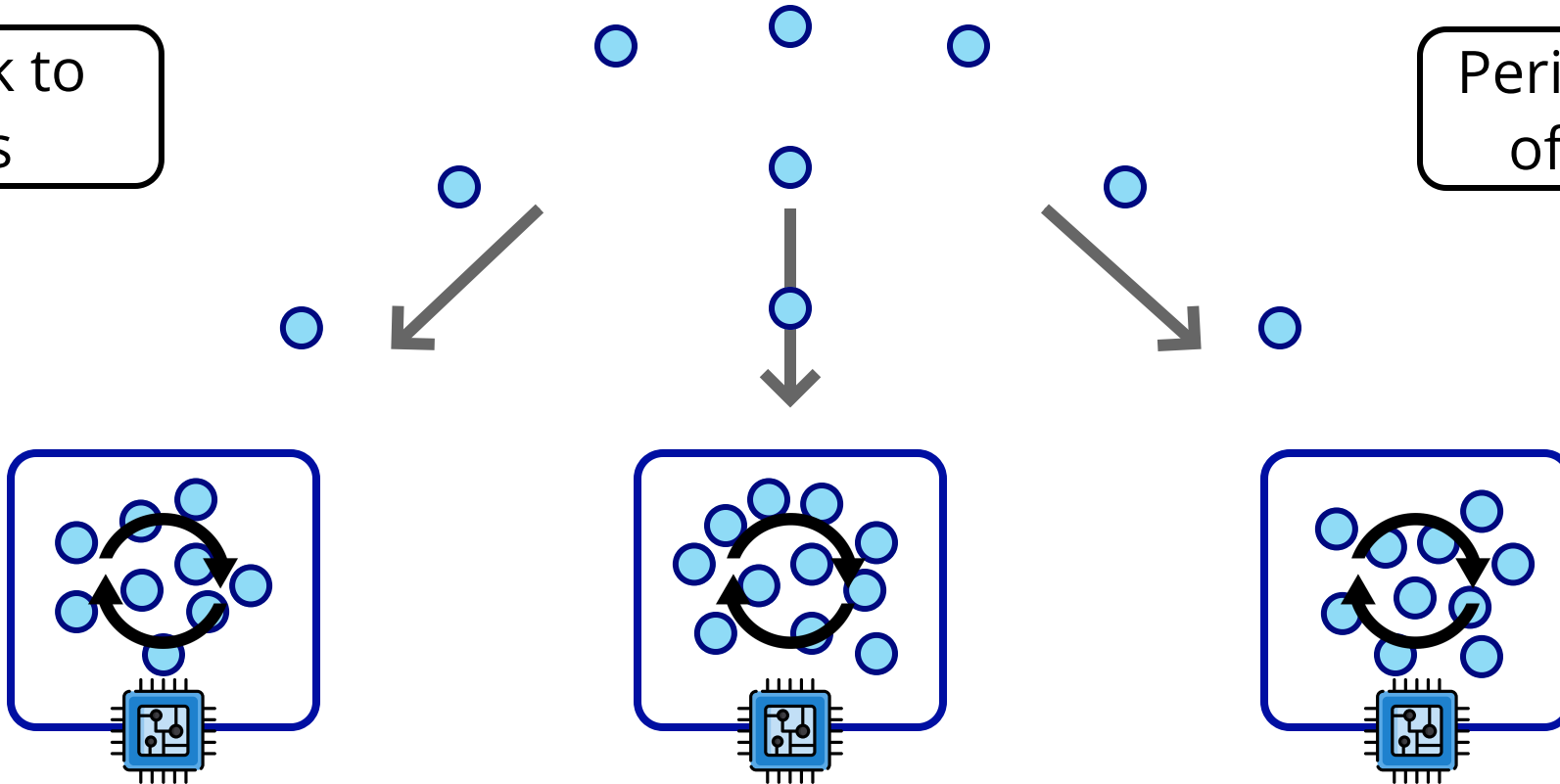


Trivial scheduler v2

Round-robin distribution

Tasks stick to
threads

Periodic update
of each task

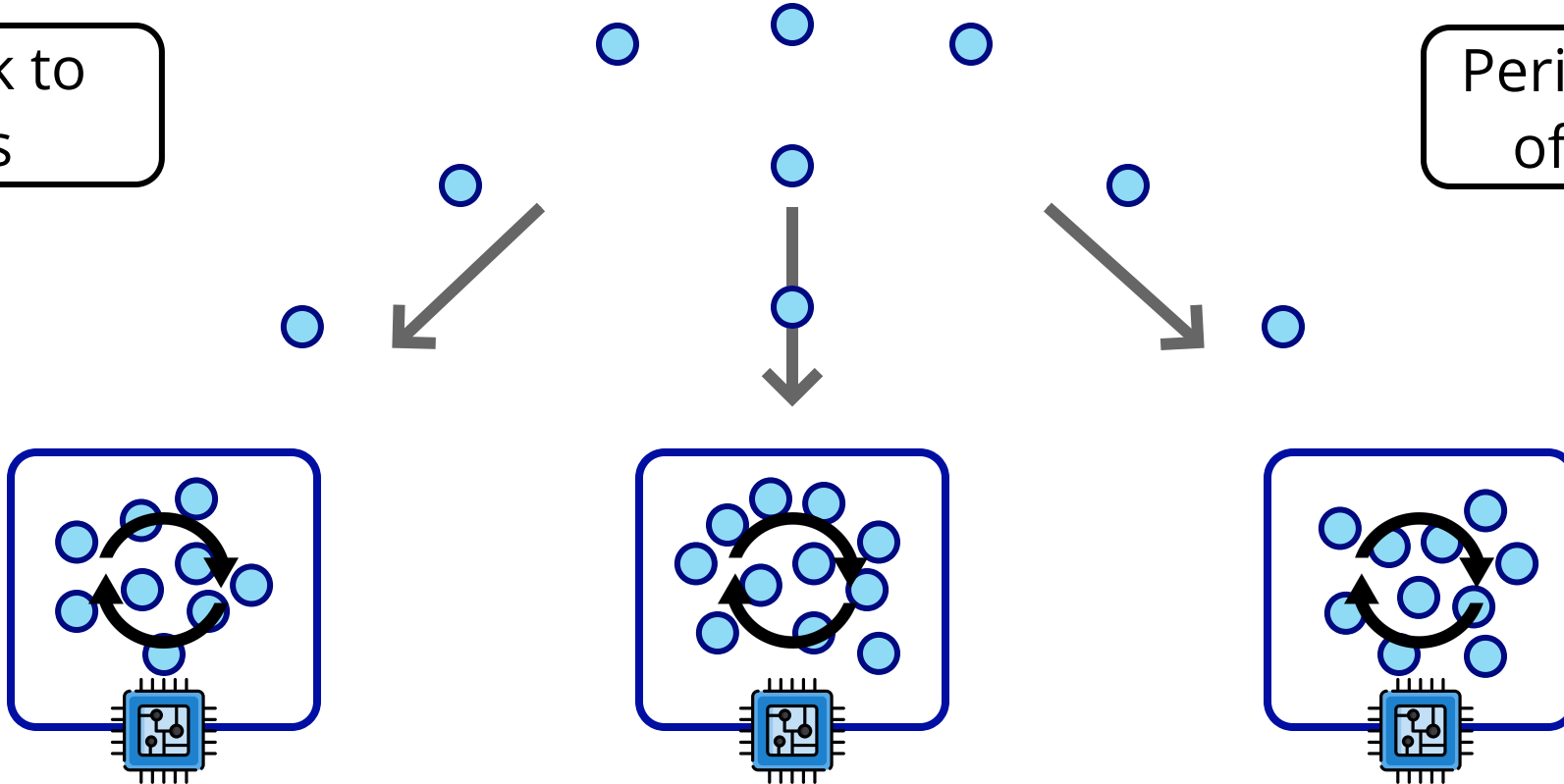


Trivial scheduler v2

Round-robin distribution

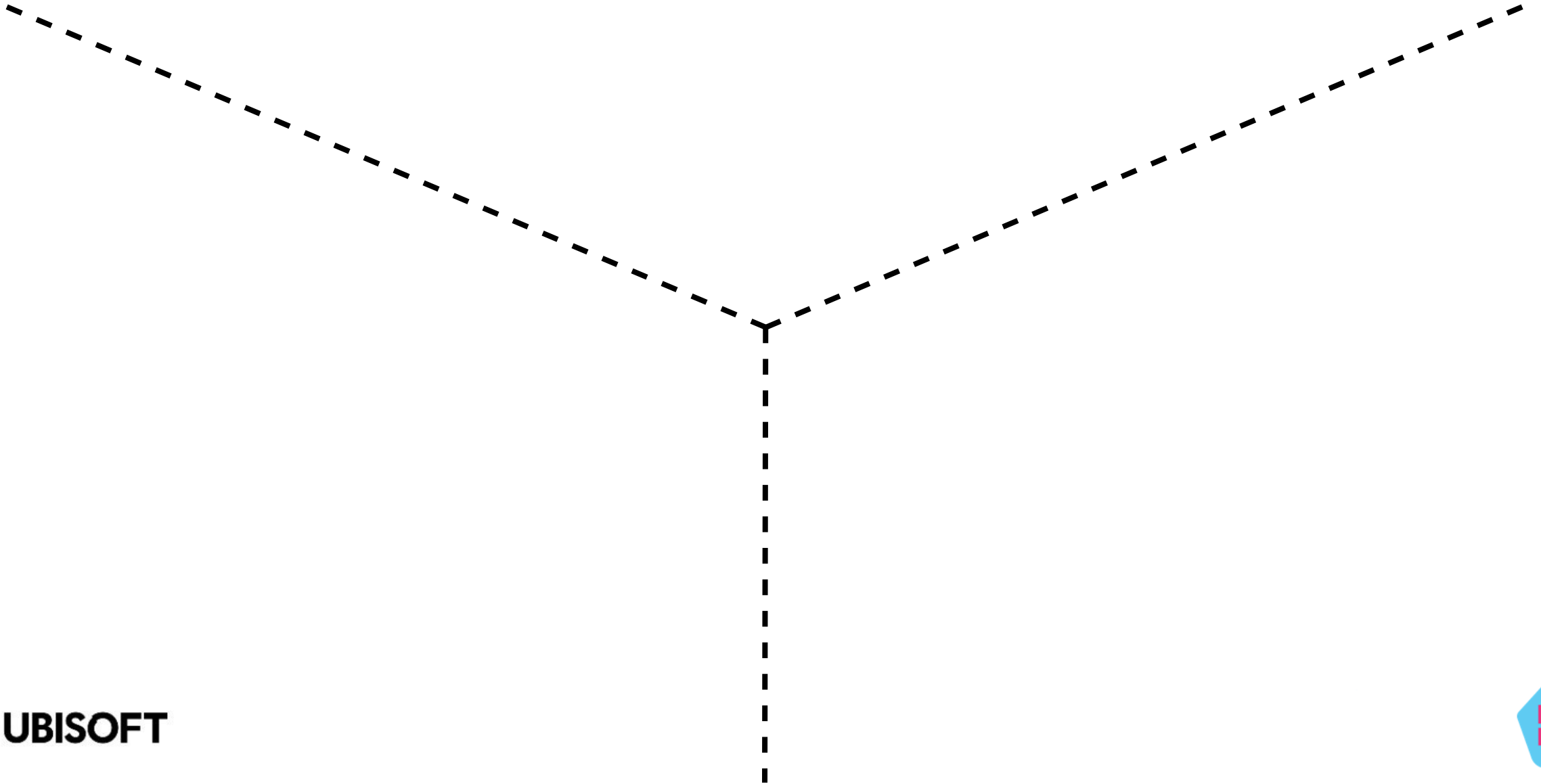
Tasks stick to
threads

Periodic update
of each task



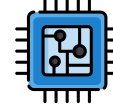
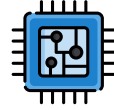
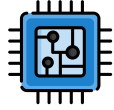
Updater "updates" all tasks all the time

Trivial scheduler v2 problems



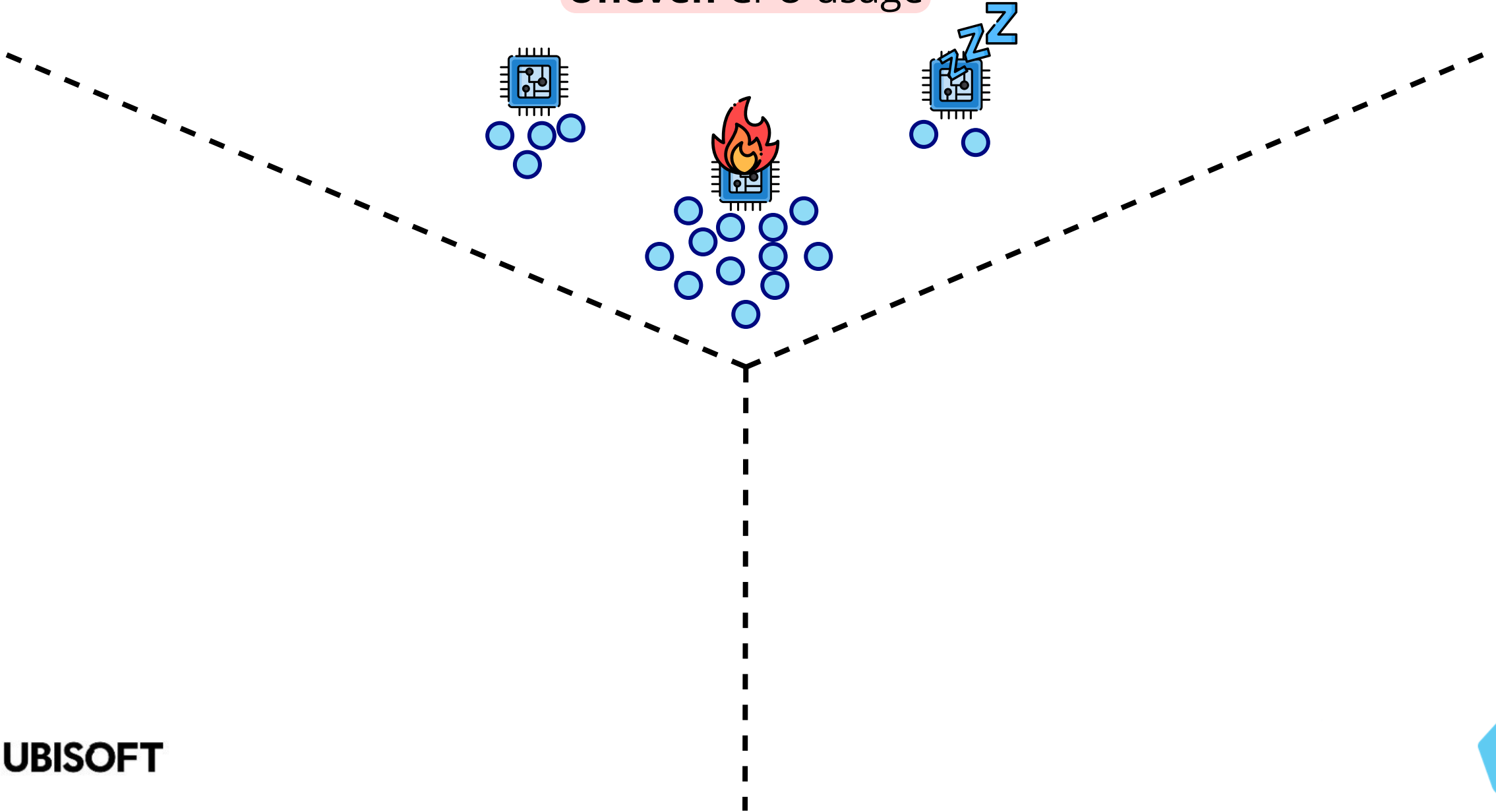
Trivial scheduler v2 problems

Uneven CPU usage



Trivial scheduler v2 problems

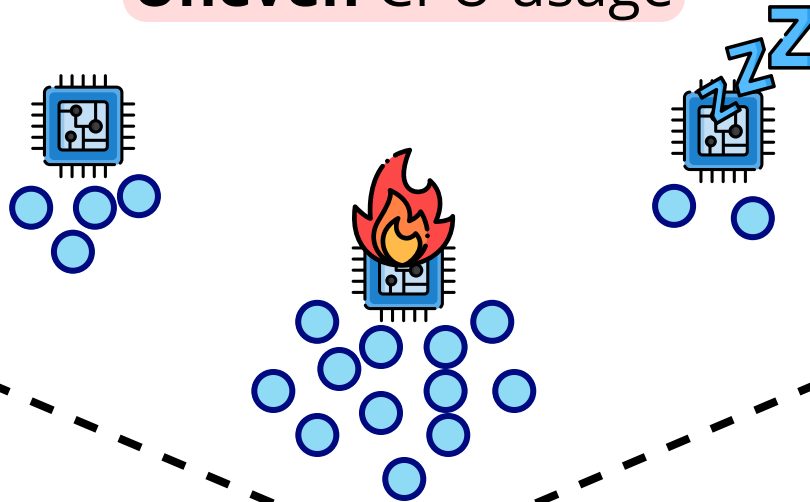
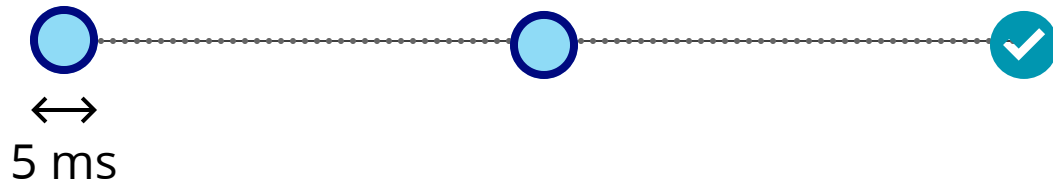
Uneven CPU usage



Trivial scheduler v2 problems

Uneven CPU usage

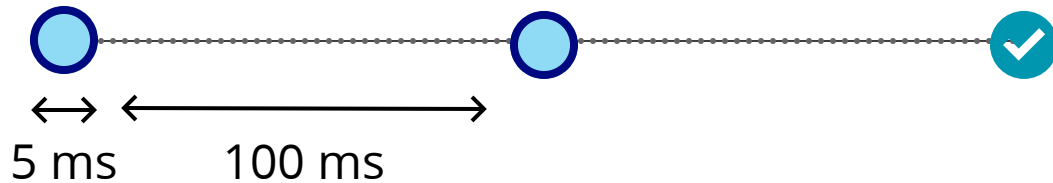
High latency



Trivial scheduler v2 problems

Uneven CPU usage

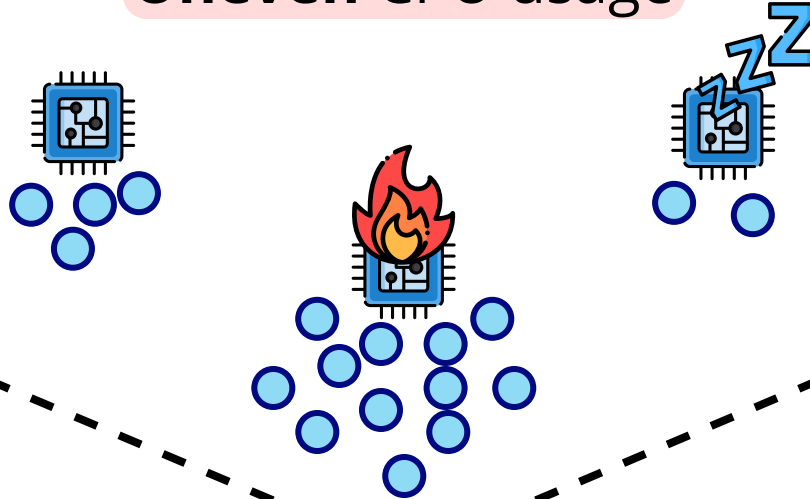
High latency



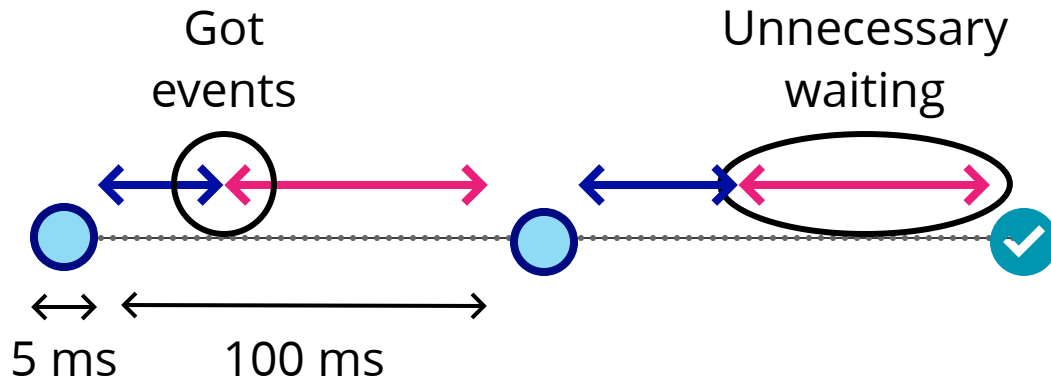
Min latency is **215 ms**

Trivial scheduler v2 problems

Uneven CPU usage



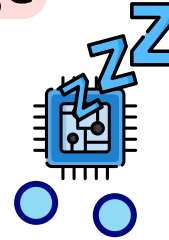
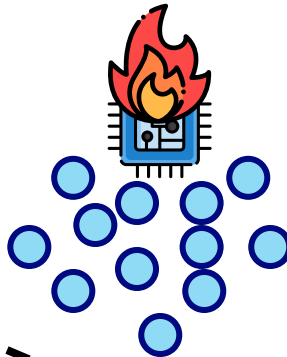
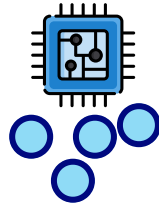
High latency



Min latency is **215 ms**

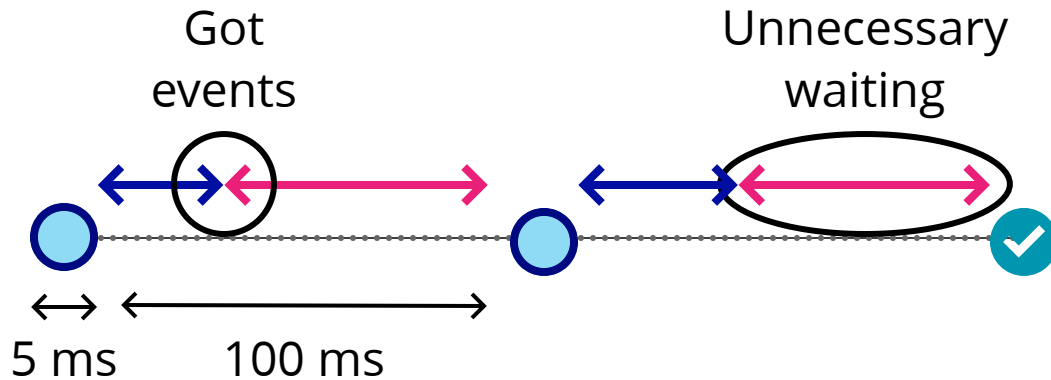
Trivial scheduler v2 problems

Uneven CPU usage

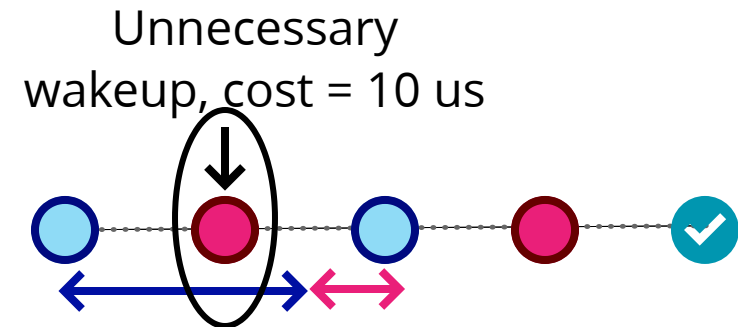


High latency

Waste of CPU



Min latency is **215 ms**



x10 000 tasks =
100 ms wasted

Summary requirements

Summary requirements

Fairness

Summary requirements

Fairness

Coroutines

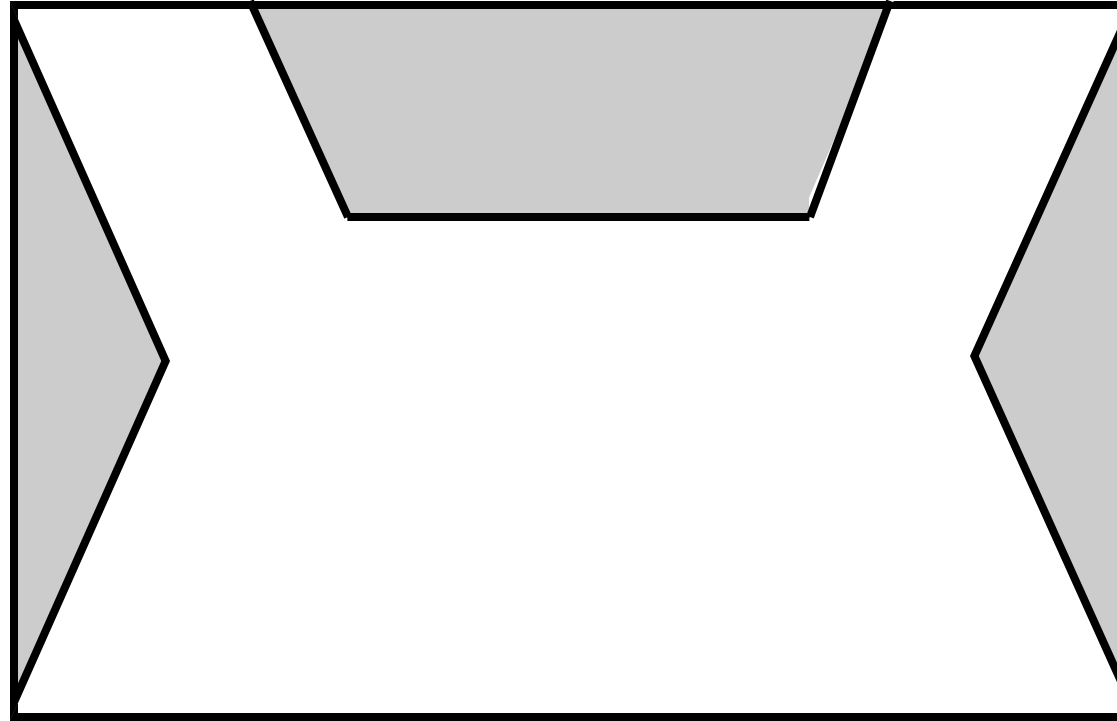
Summary requirements

Fairness

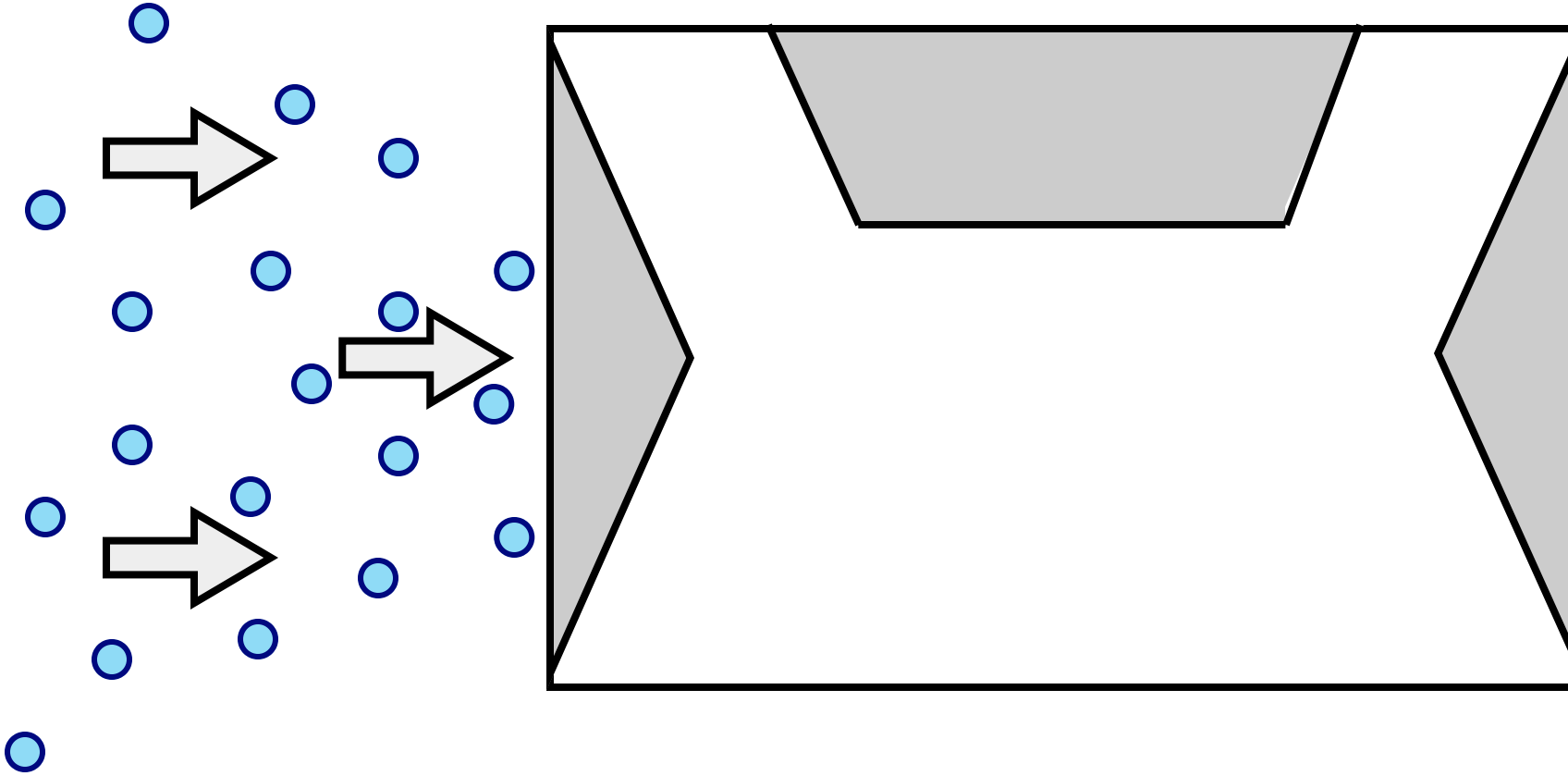
Coroutines

Events

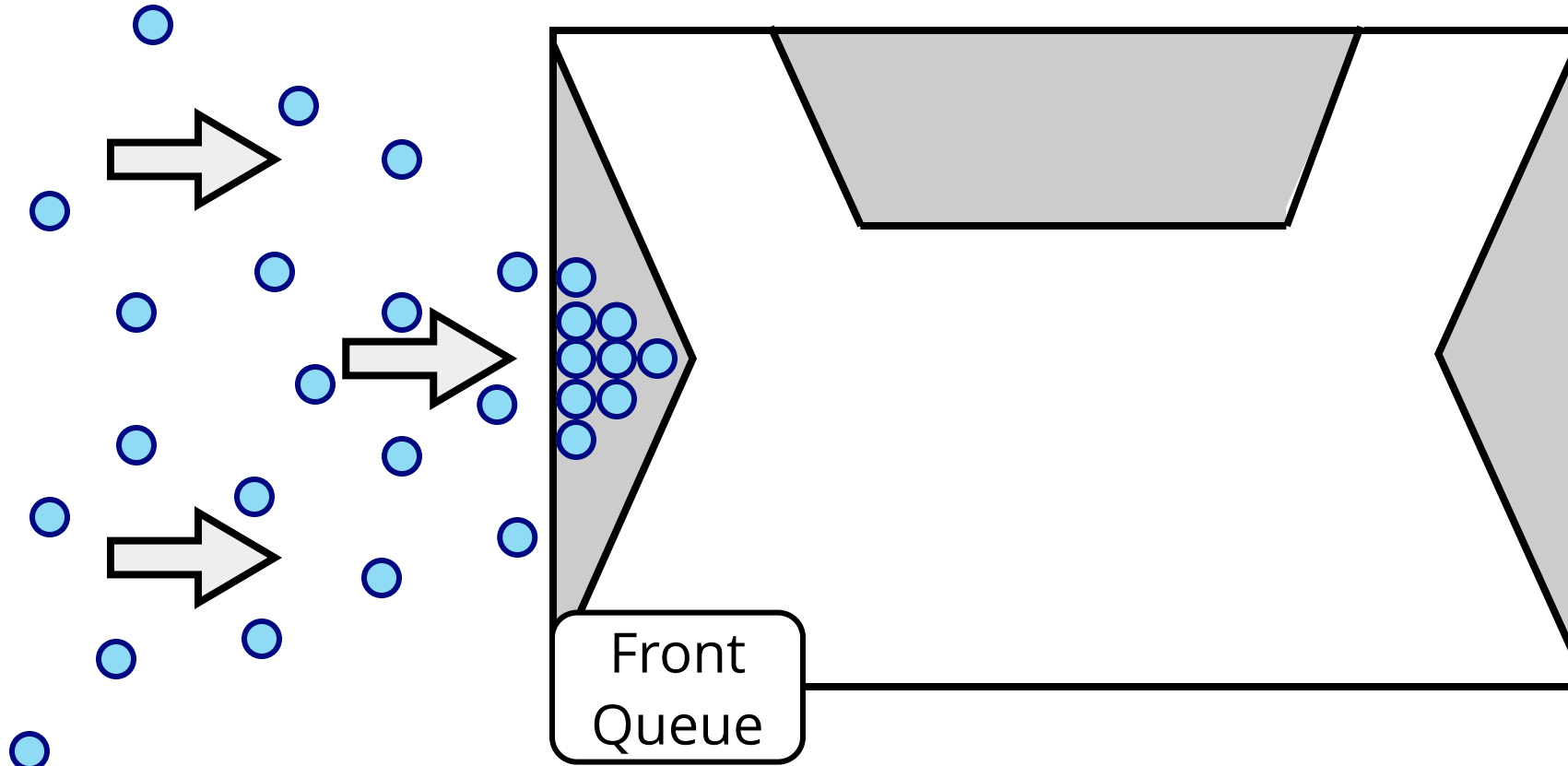
TaskScheduler



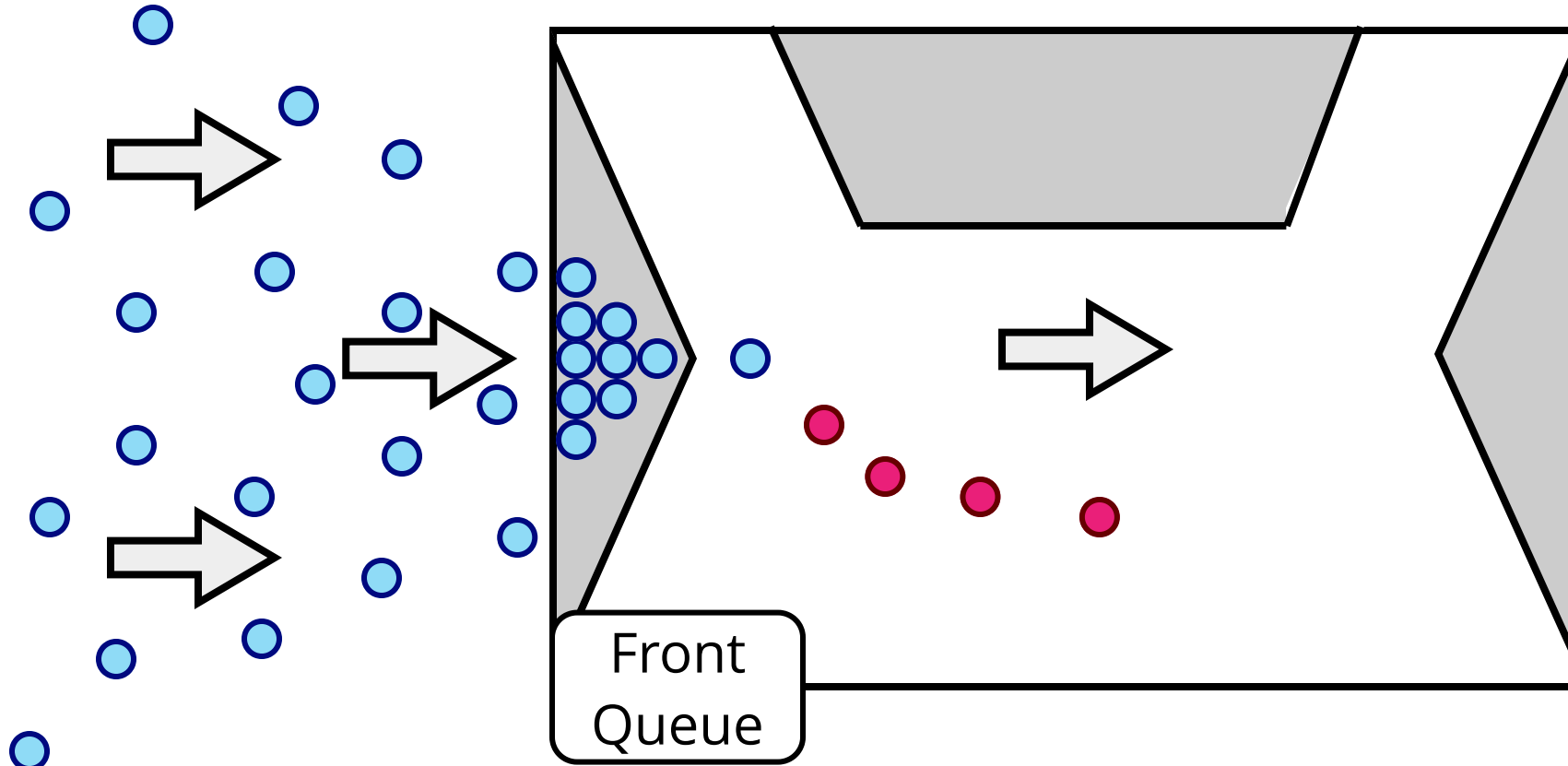
TaskScheduler



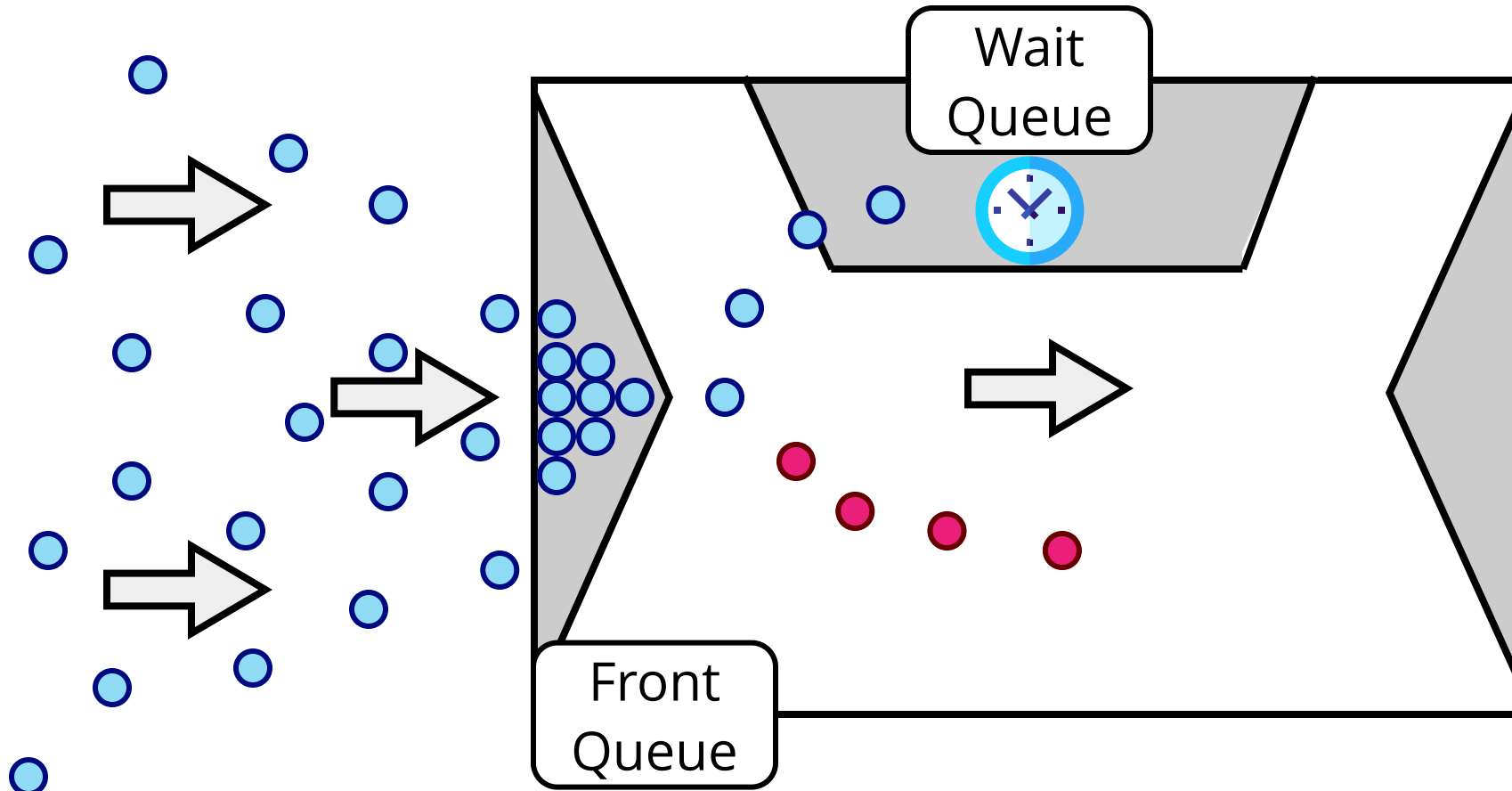
TaskScheduler



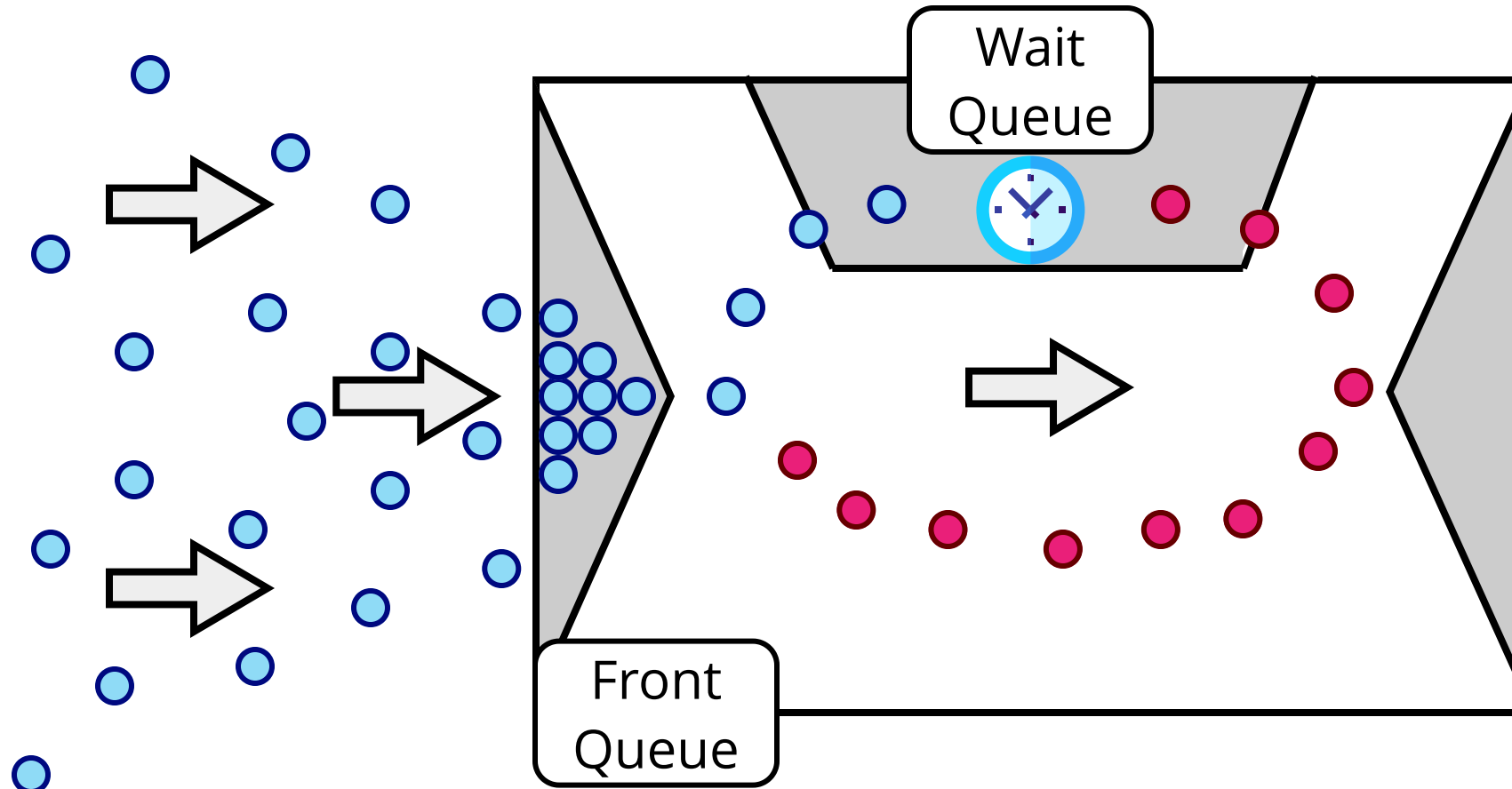
TaskScheduler



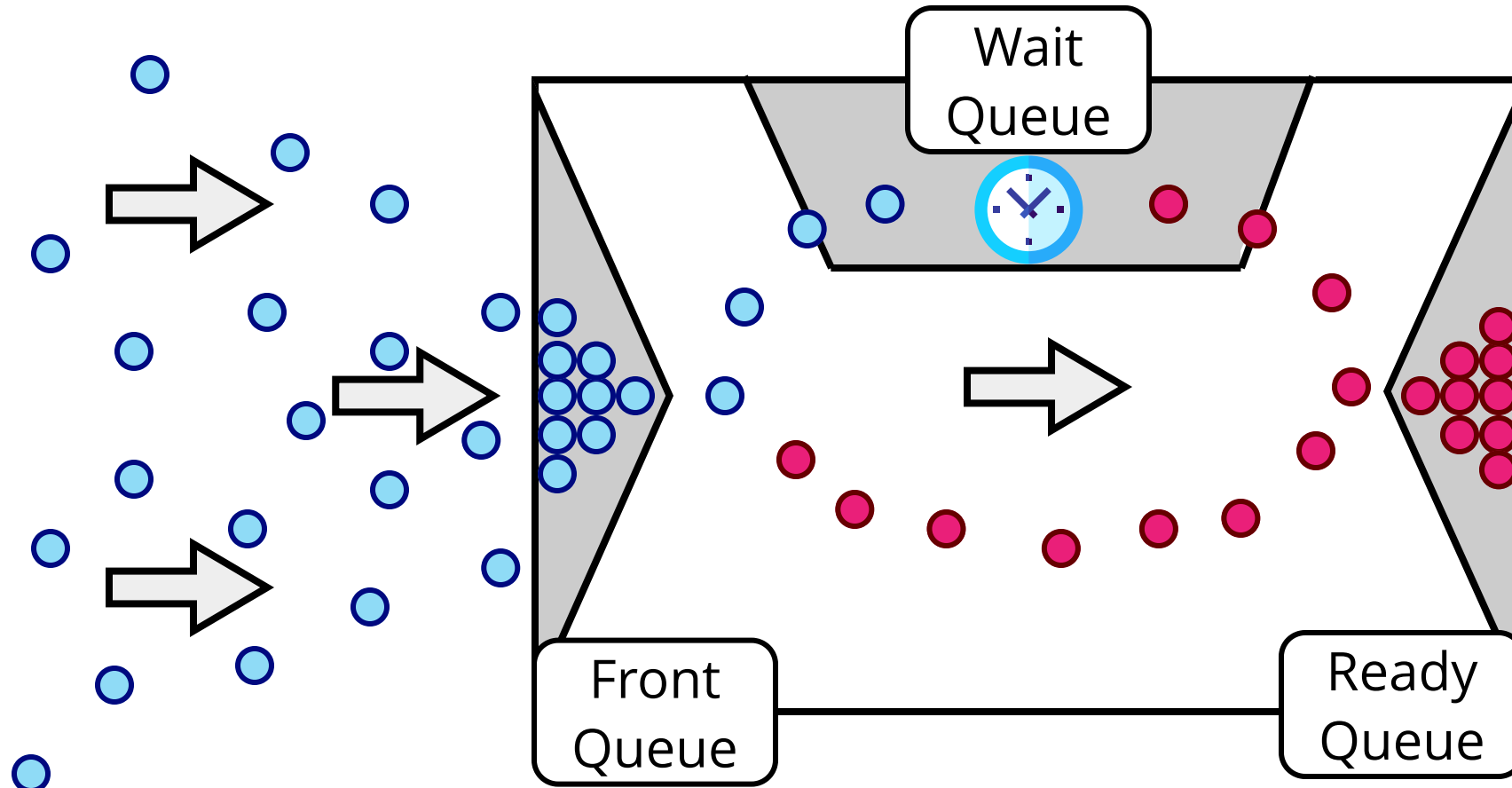
TaskScheduler



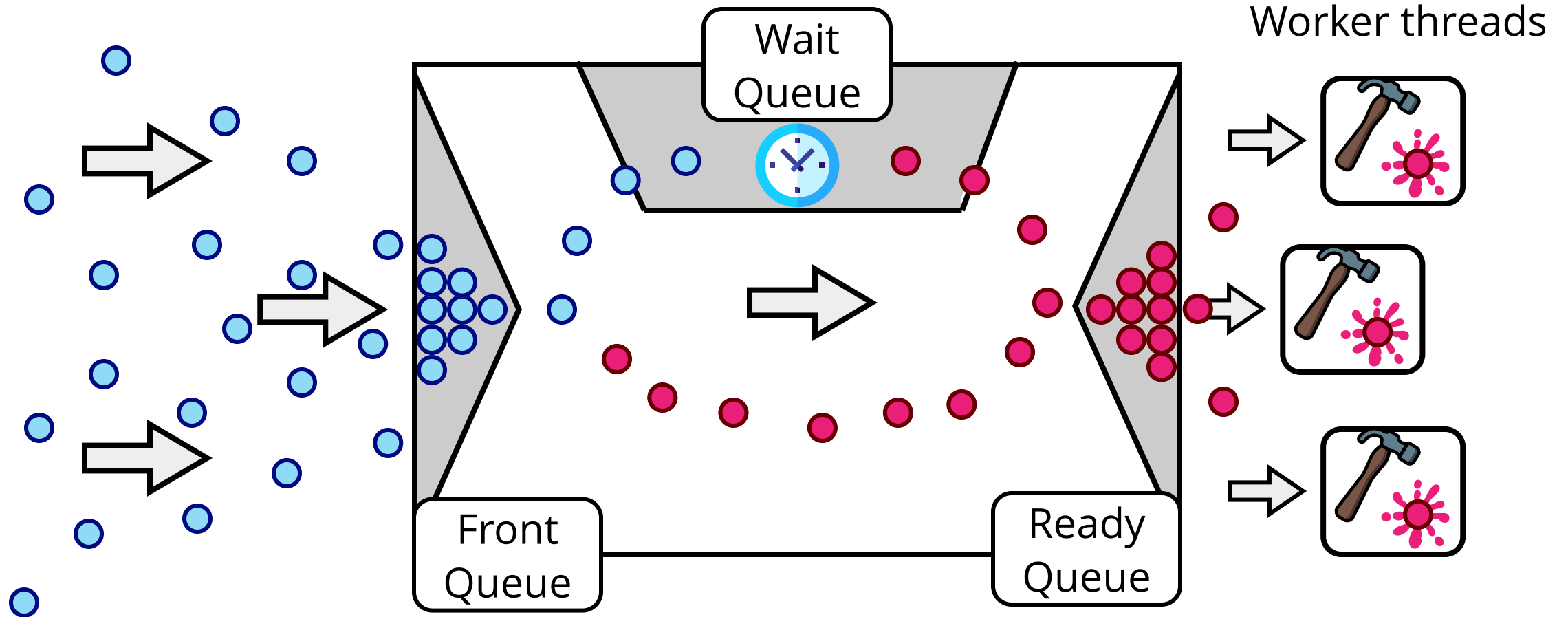
TaskScheduler



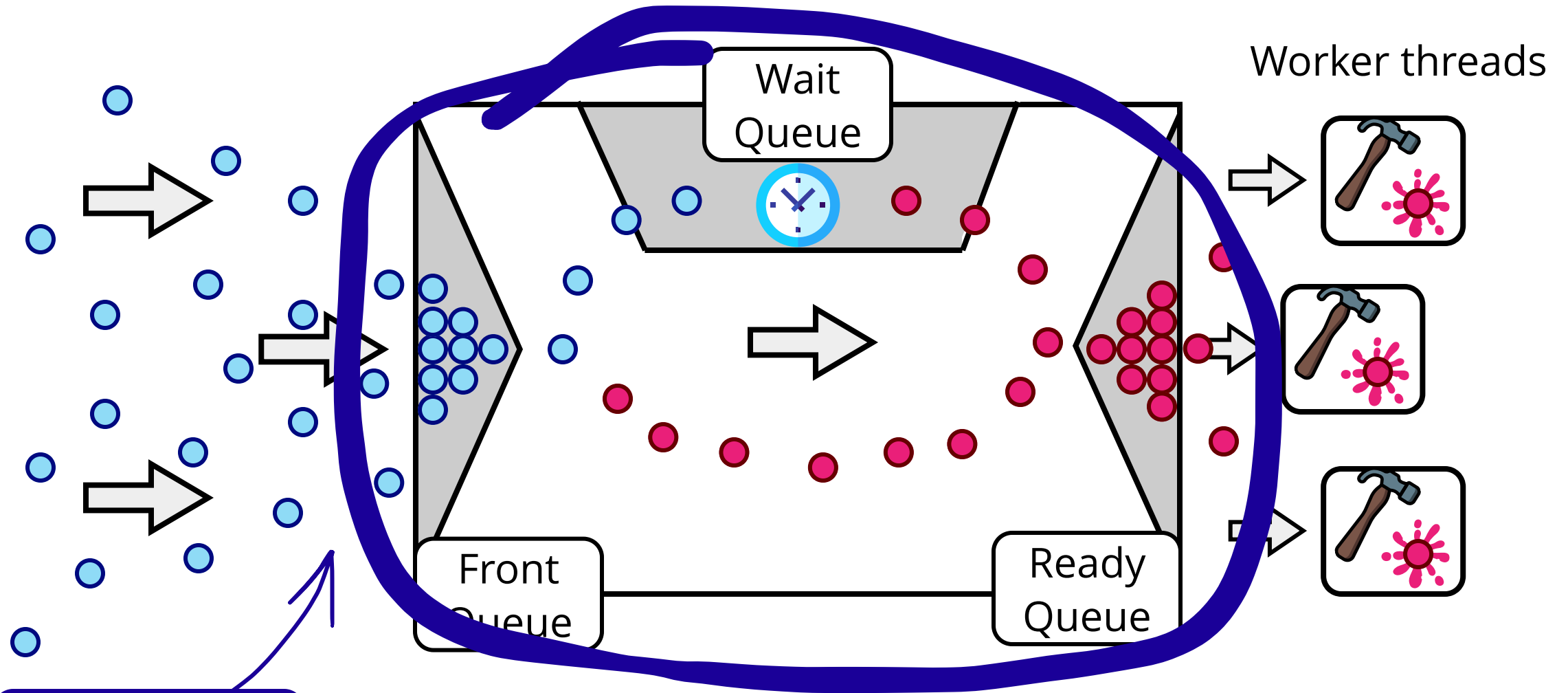
TaskScheduler



TaskScheduler

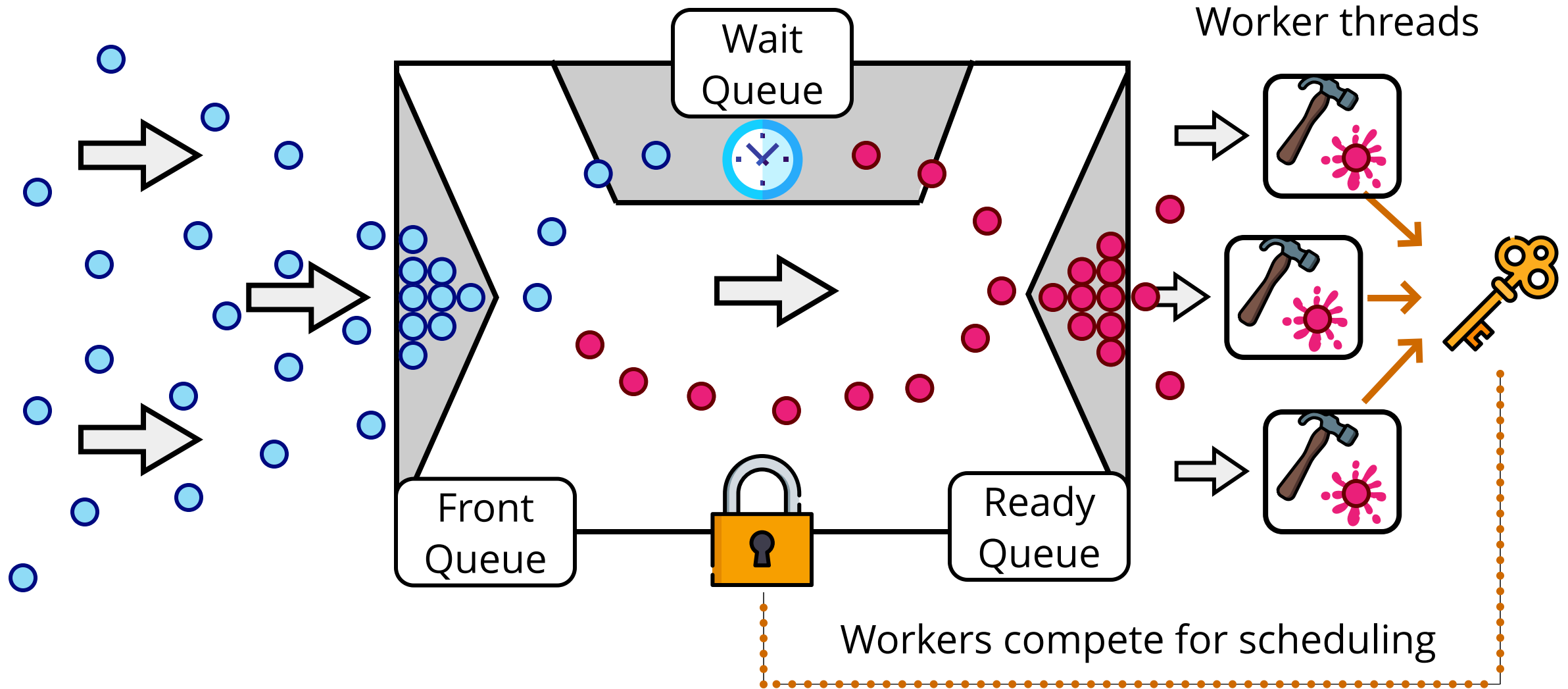


TaskScheduler

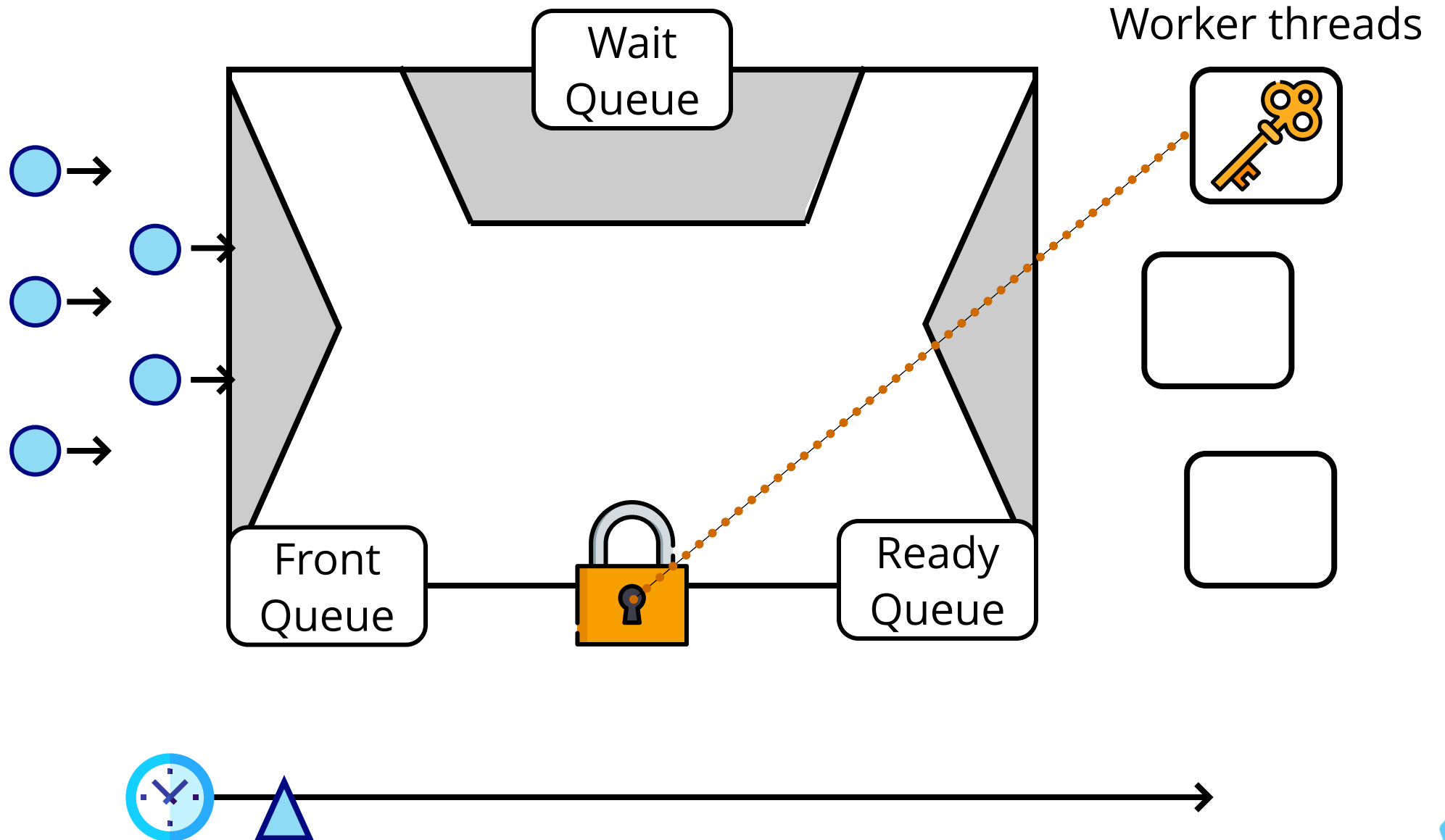


Who processes the queues?

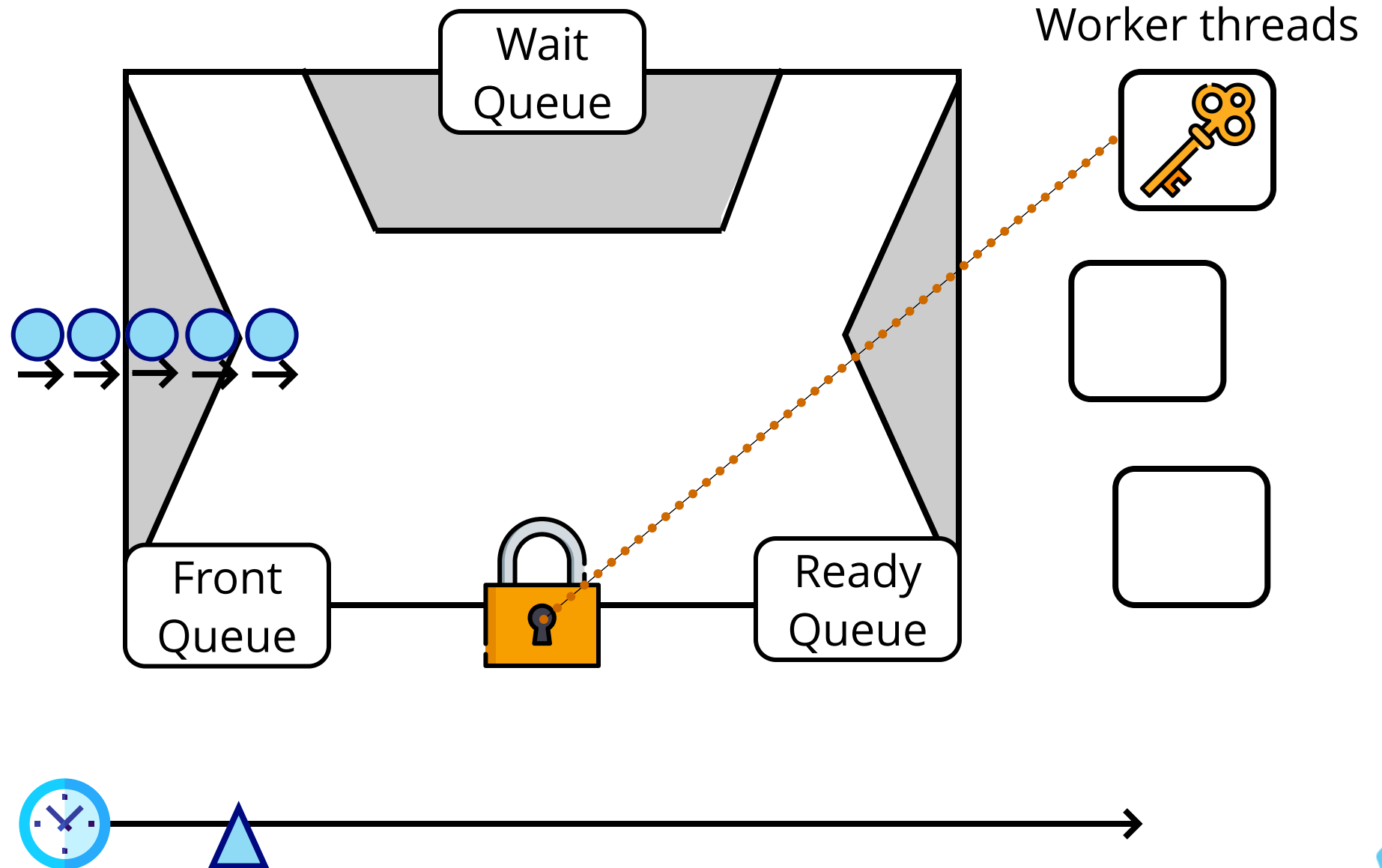
TaskScheduler



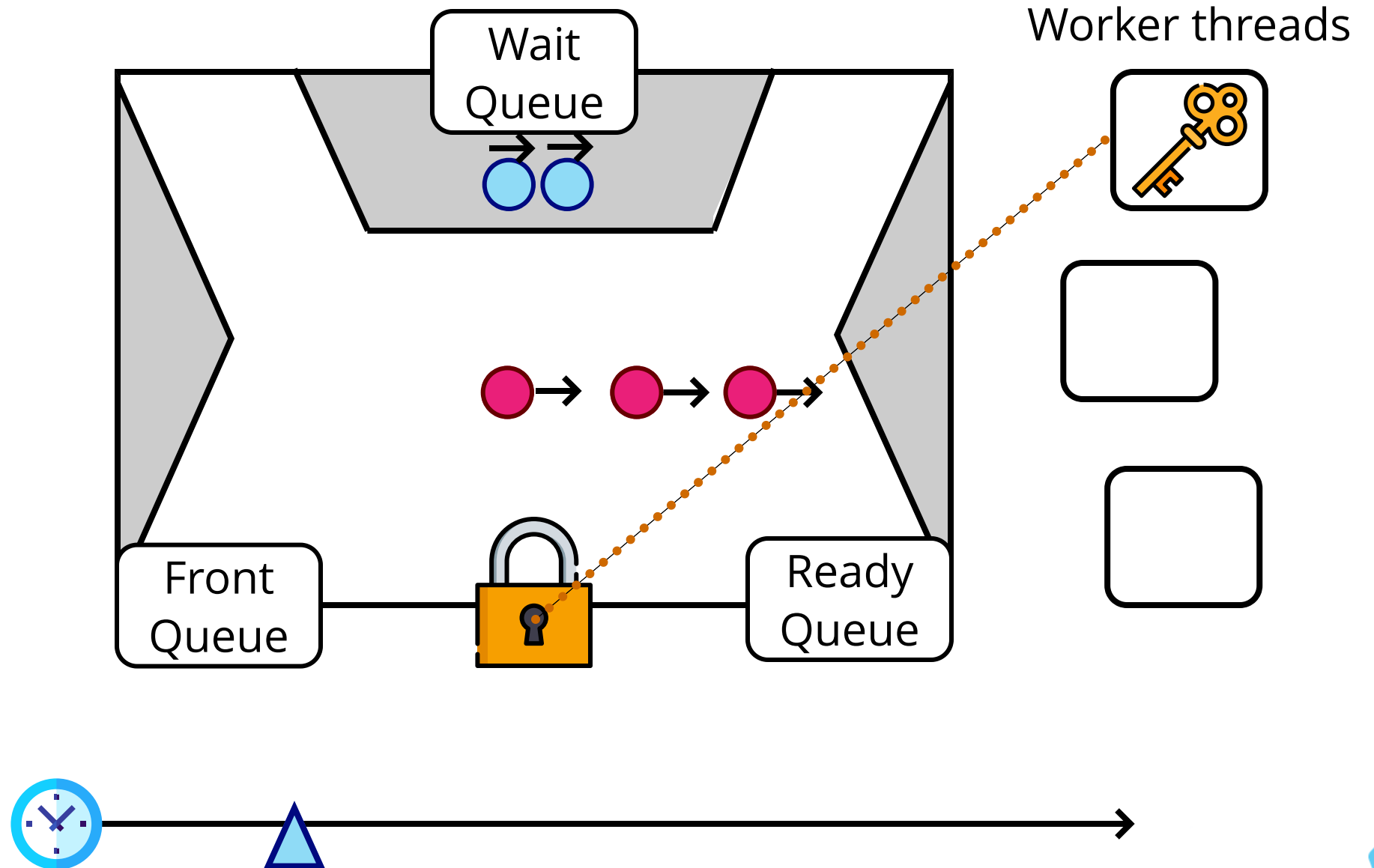
TaskScheduler example



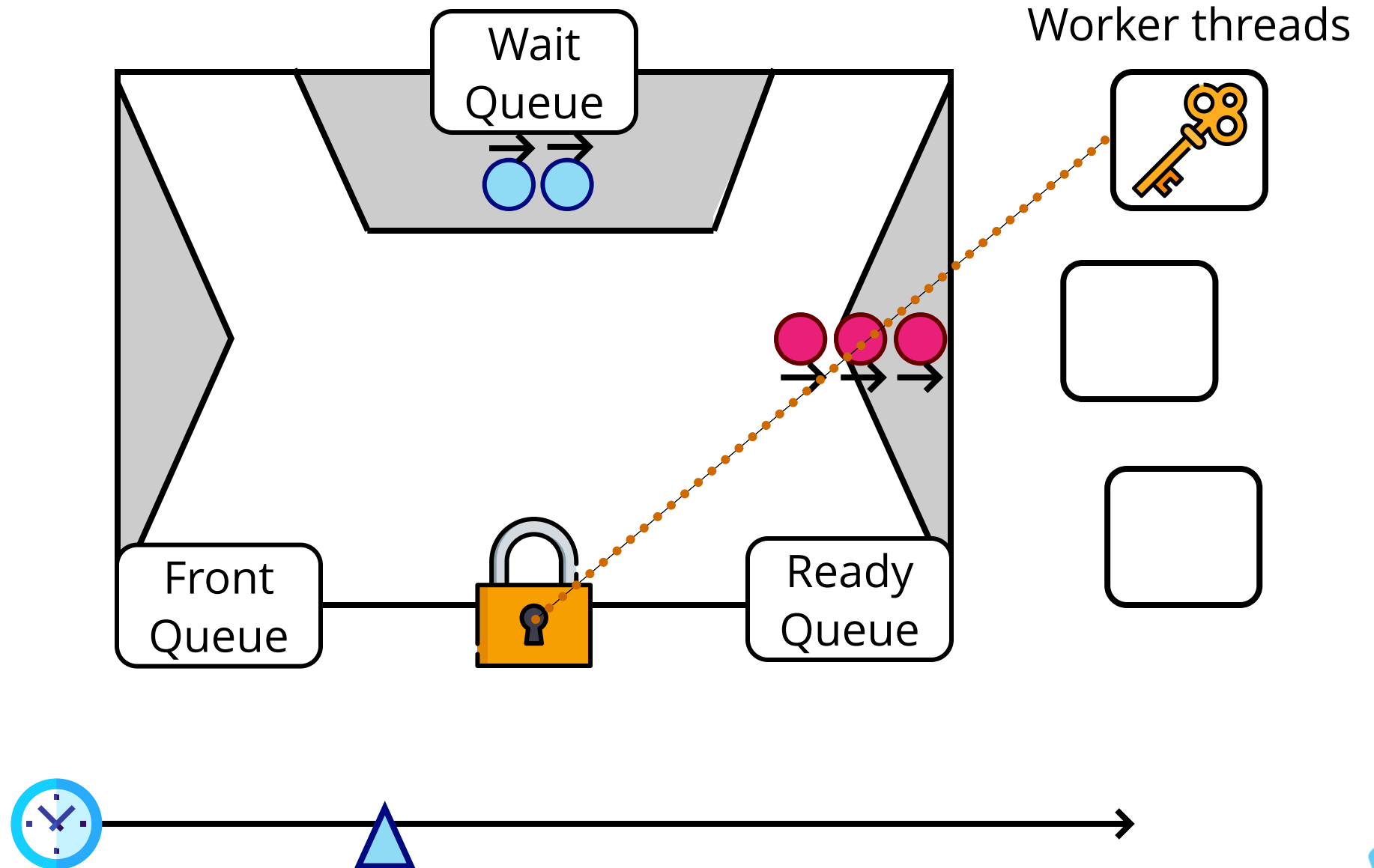
TaskScheduler



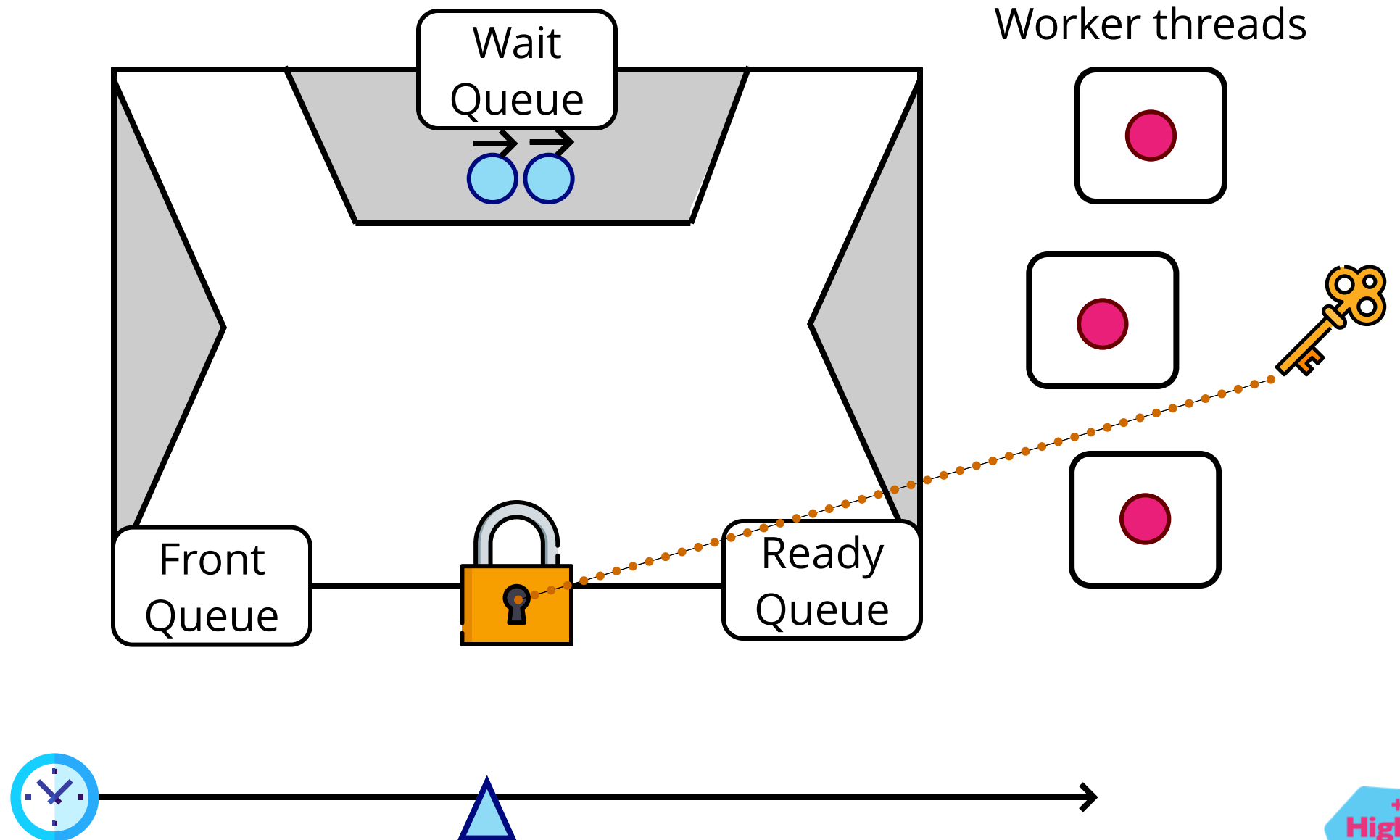
TaskScheduler



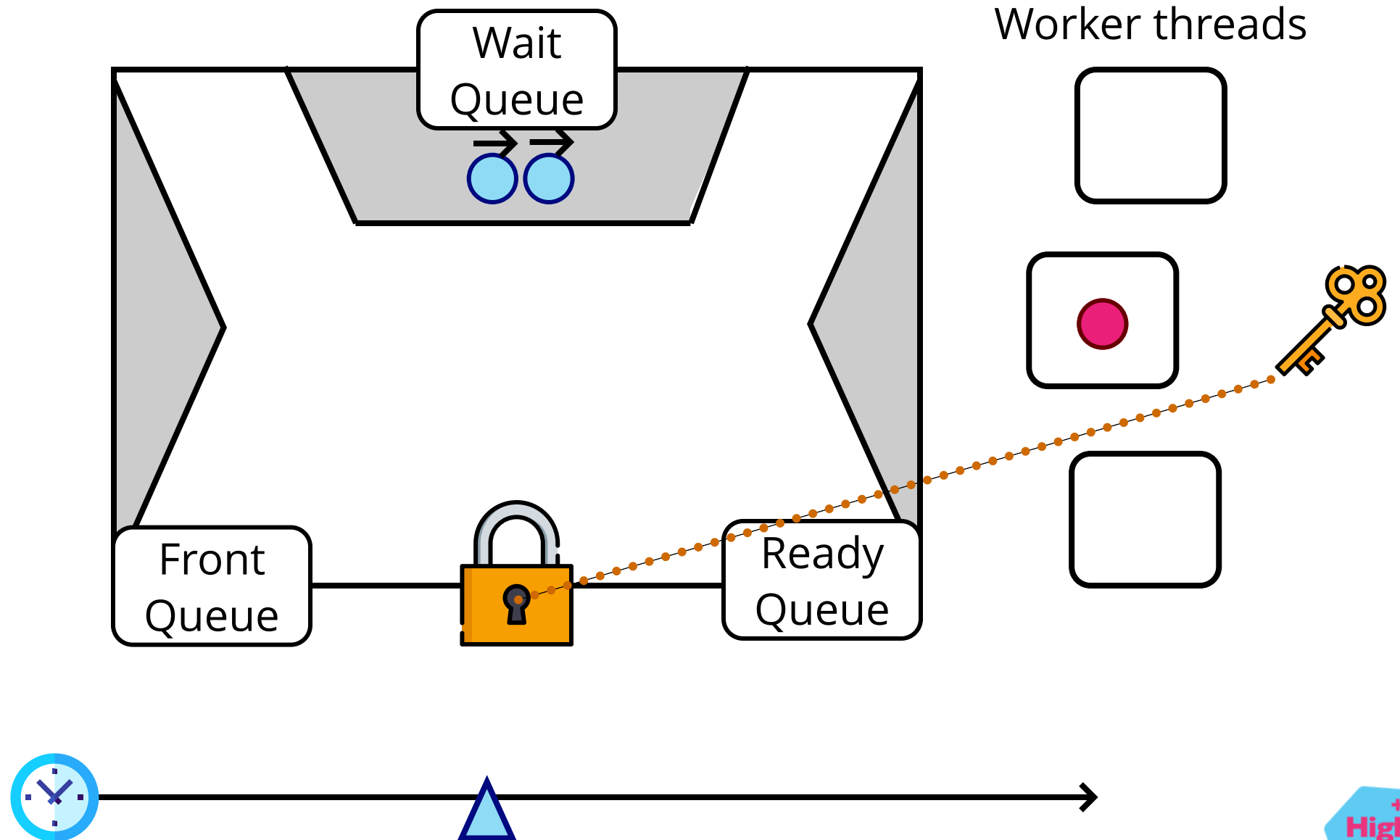
TaskScheduler



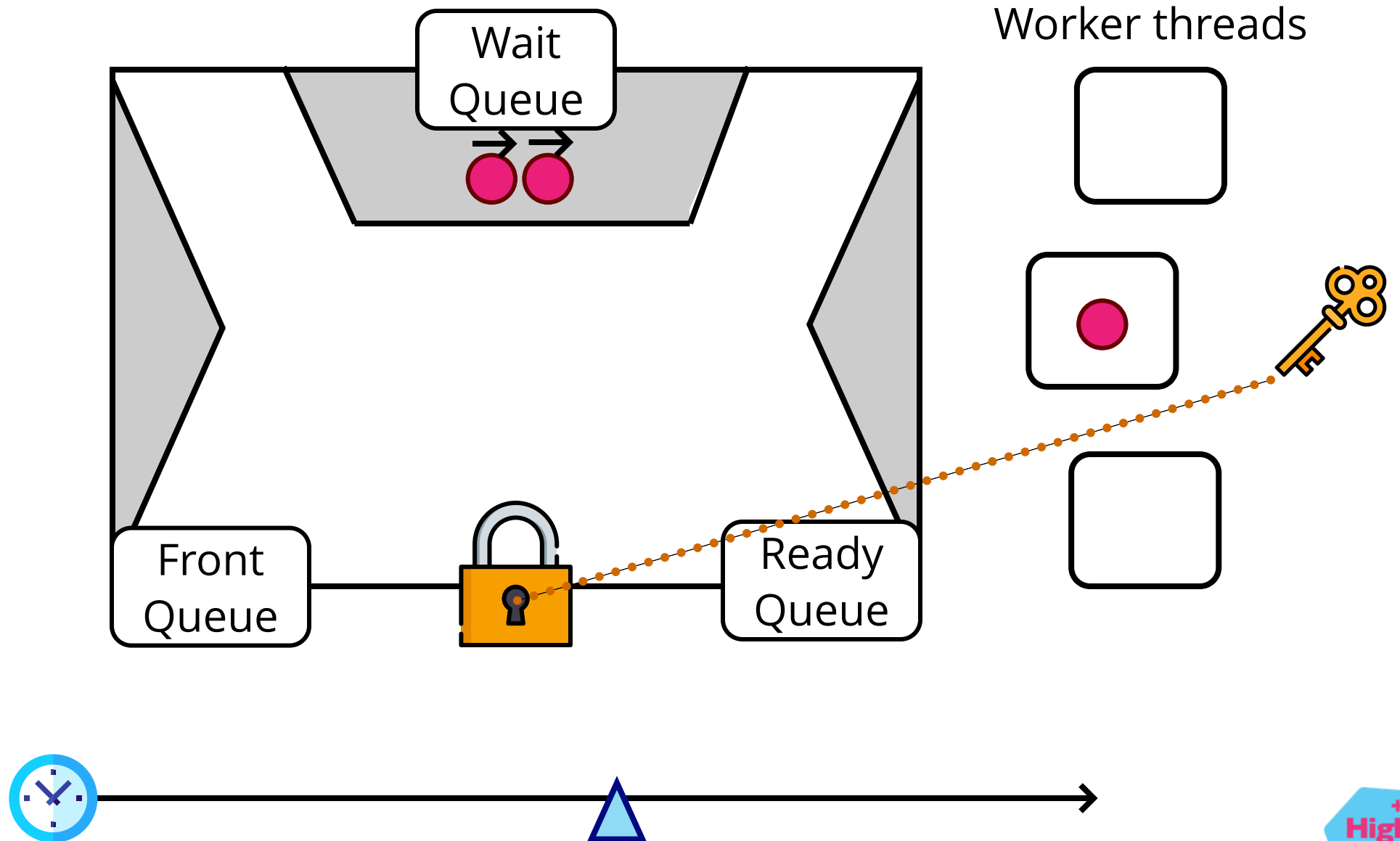
TaskScheduler



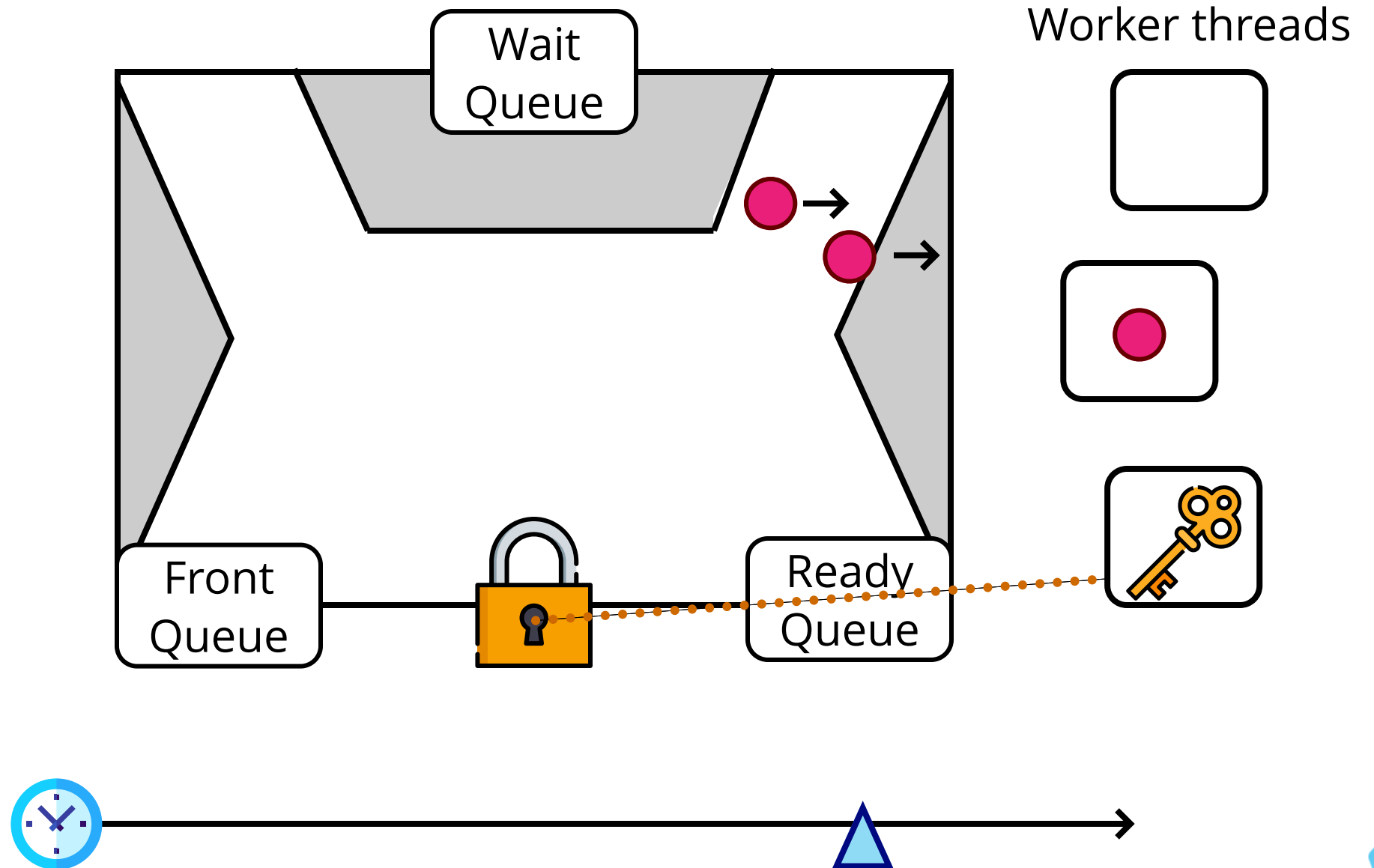
TaskScheduler



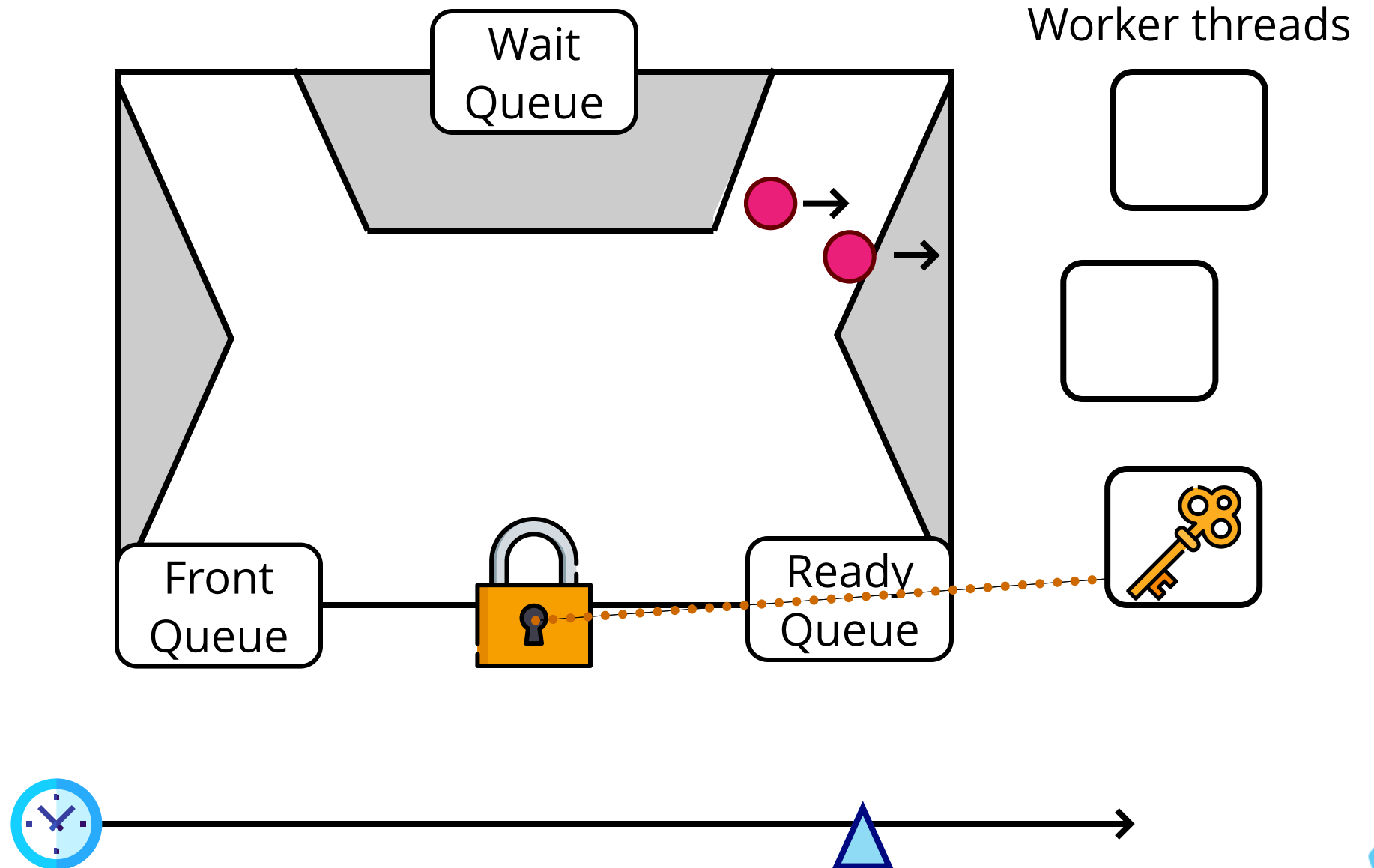
TaskScheduler



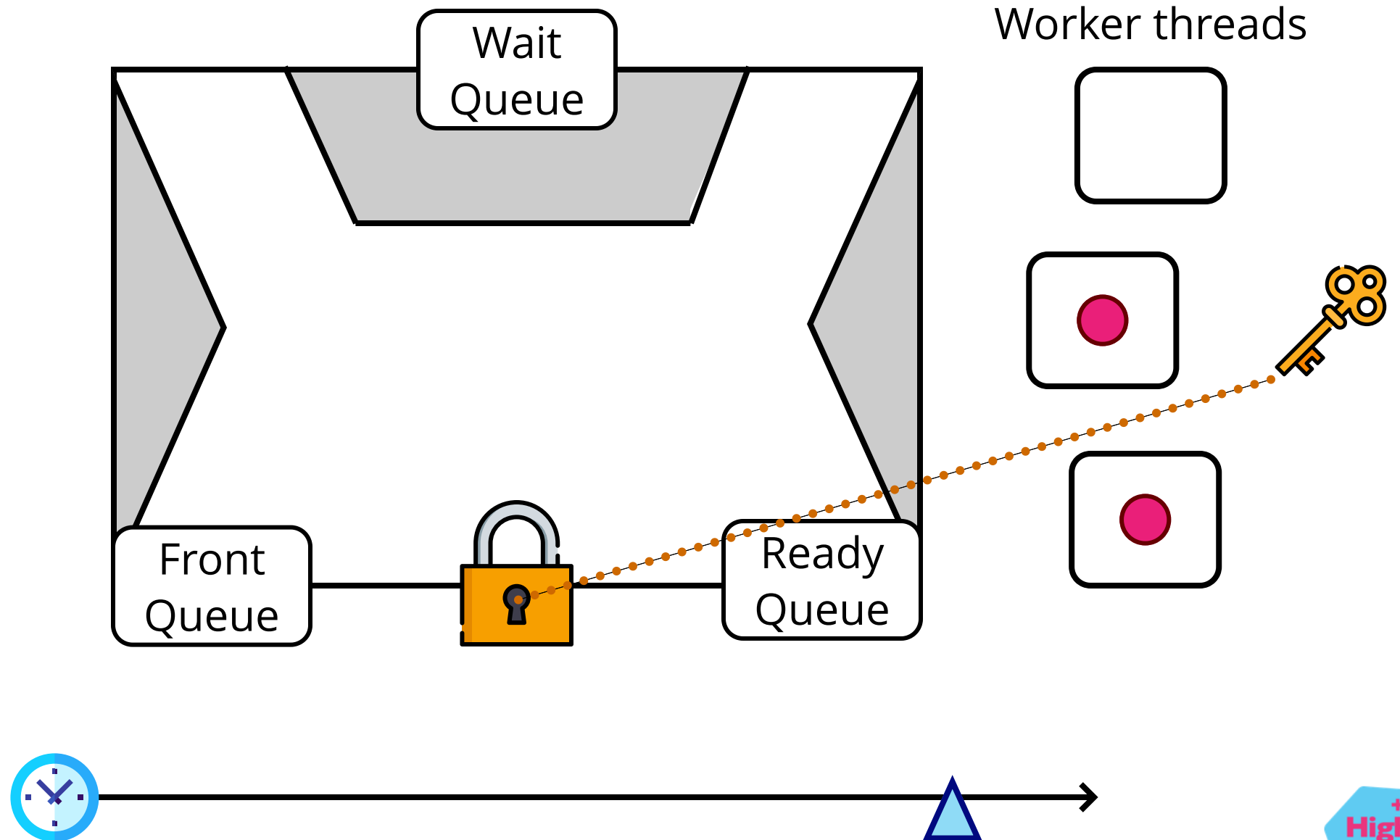
TaskScheduler



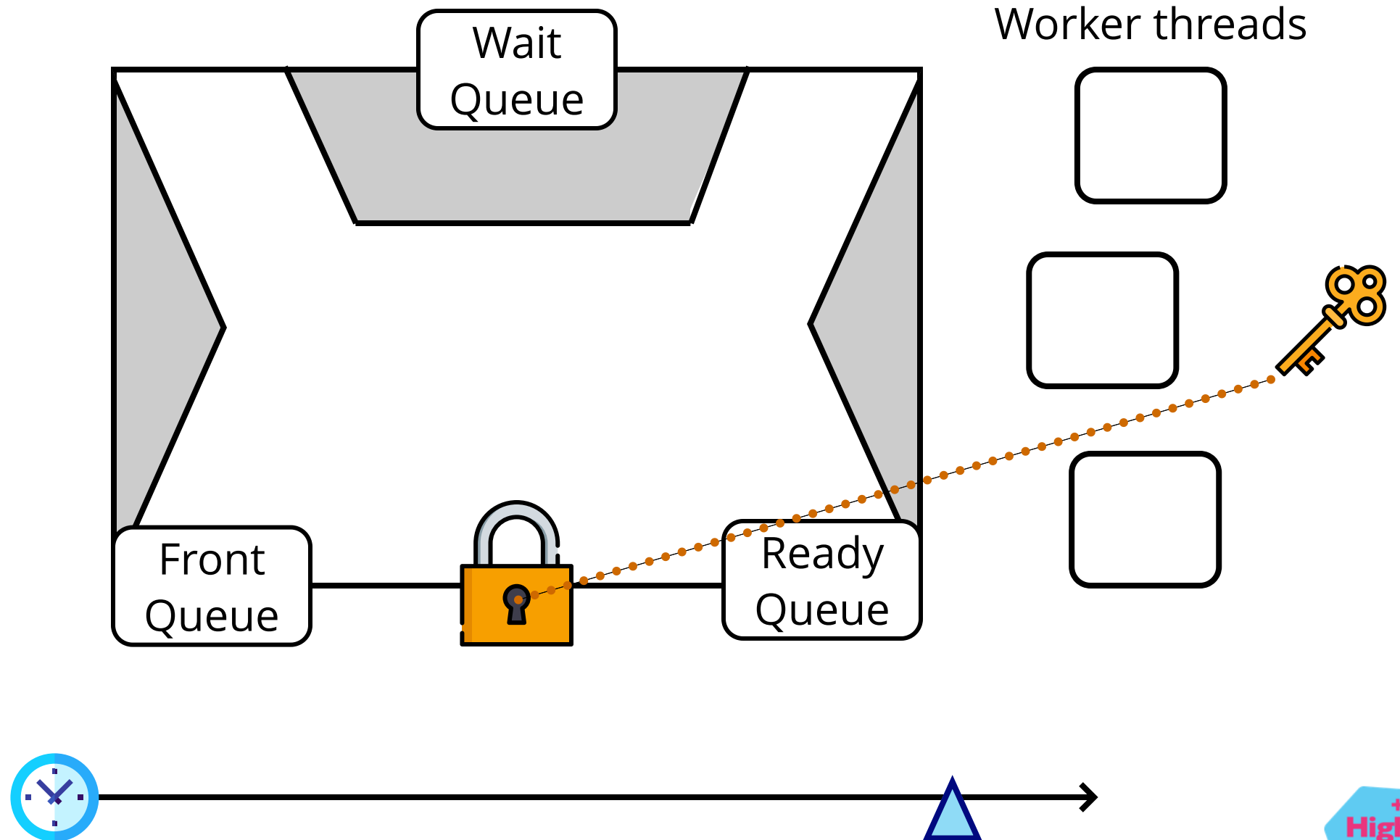
TaskScheduler



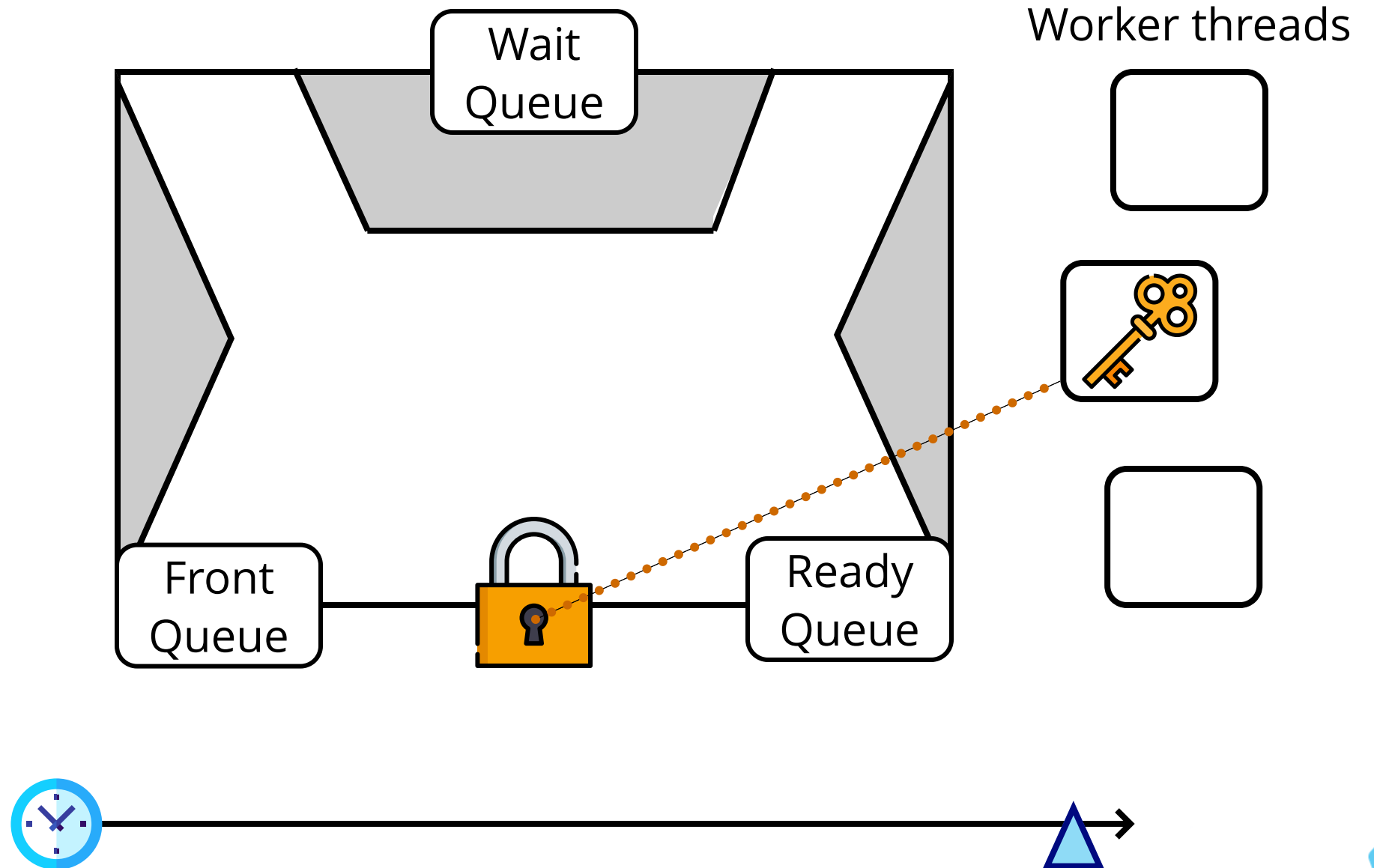
TaskScheduler



TaskScheduler



TaskScheduler



Dependencies

Mutex

Containers

Condition variable

Lock-free atomics

Lock-free atomics

Atomically get the old value

```
AtomicLoad(var)
{
    return var;
}
```

* Google "**memory models**"
about why it has to be atomic

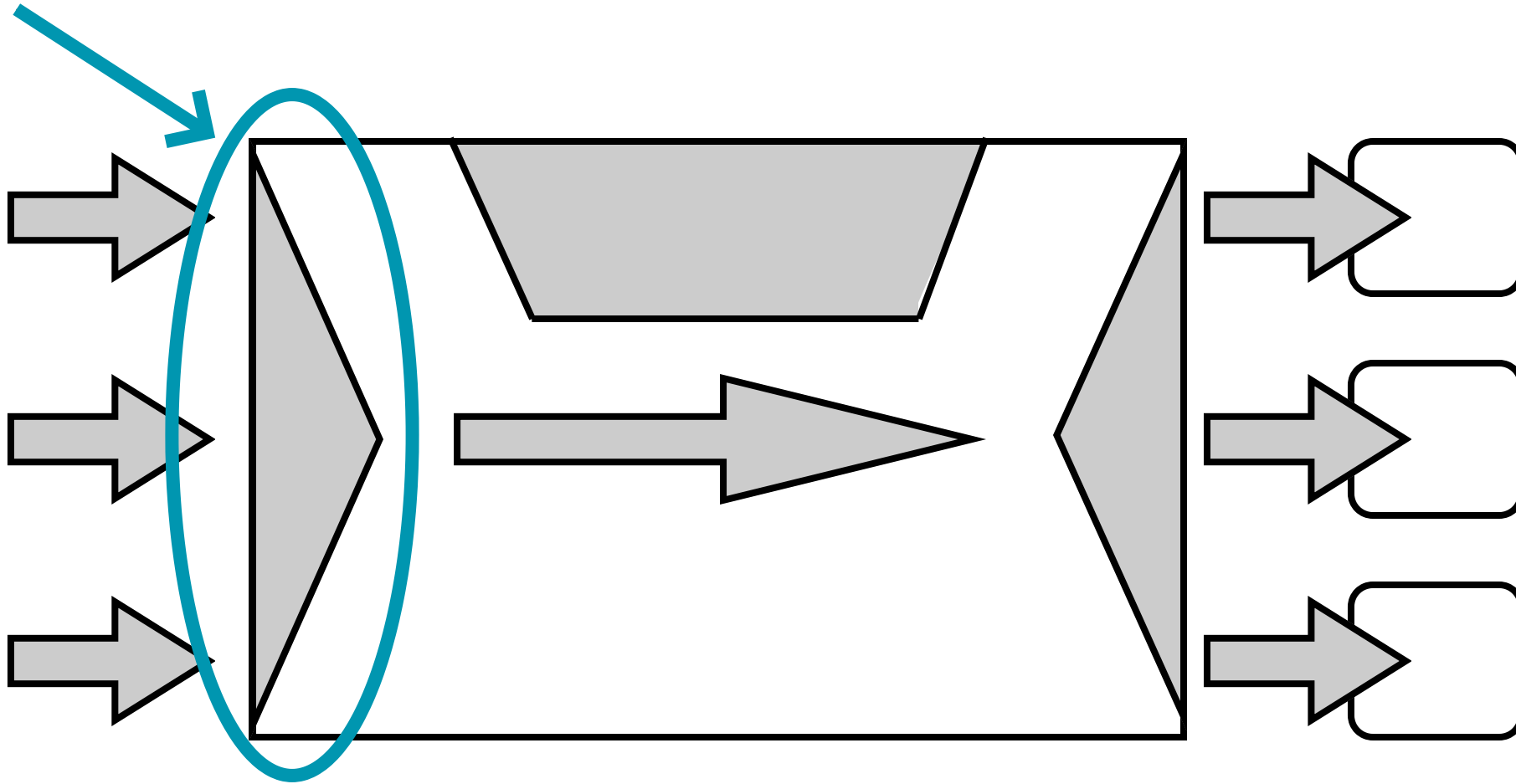
Atomically set if the old value equals something

```
AtomicCompareExchange(var, new_value, check)
{
    if (var != check)
        return false;
    var = new_value;
    return true;
}
```

Atomically set a new value and get the old value

```
AtomicExchange(var, new_value)
{
    old_value = var;
    var = new_value;
    return old_value;
}
```

Front queue



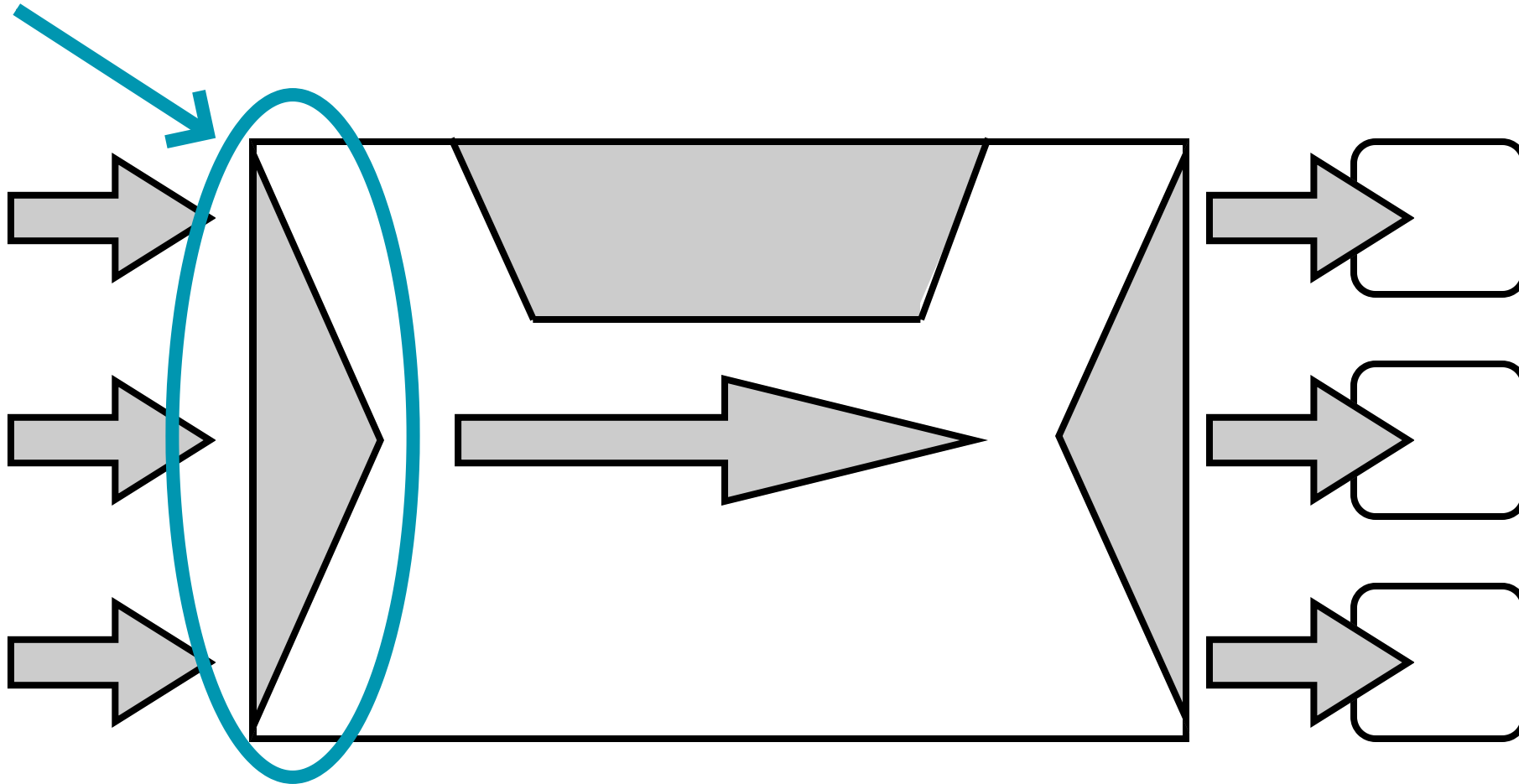
Front queue

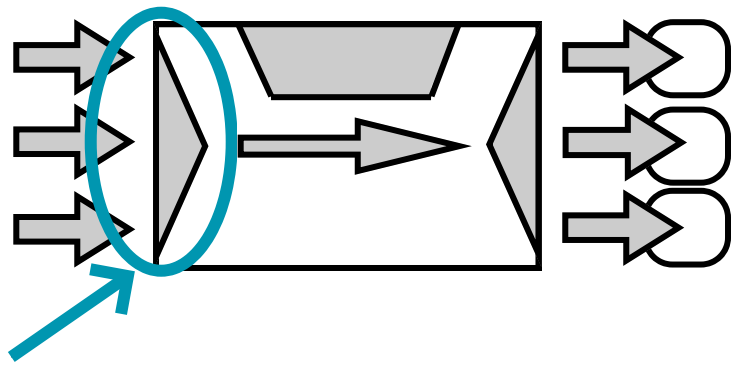
Multi-Producer-Single-Consumer*

■ All threads ■

■ Sched-worker ■

* Common notation for queues:
MPSC, MPMC, SPSC, SPMC





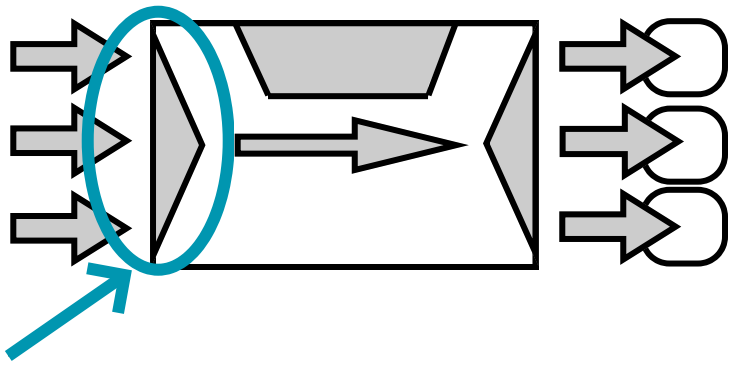
Front queue

Multi-Producer-Single-Consumer

■ ————— ■
All threads

■ ————— ■
Sched-worker

High
contention



Front queue

Multi-Producer-Single-Consumer

All threads

Sched-worker

High
contention

```
class NormalQueue:
```

```
    T* myHead;
```

```
    T* myTail;
```

```
NormalQueue::Push(T* aItem)
```

```
{
```

```
    if (myHead == nullptr)
```

```
        myHead = aItem;
```

```
    else
```

```
        myTail->myNext = aItem;
```

```
    myTail = aItem;
```

```
}
```

```
NormalQueue::Pop()
```

```
{
```

```
    if (myHead == nullptr)
```

```
        return nullptr;
```

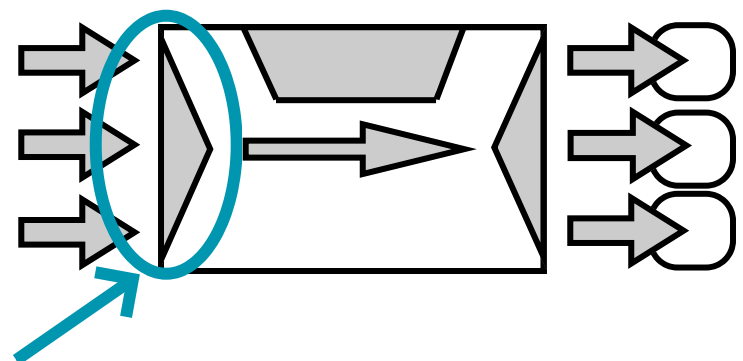
```
    T* res = myHead;
```

```
    myHead = res->myNext;
```

```
    return res;
```

```
}
```

Normally a queue
needs 2 members:
head and **tail**



Front queue

Multi-Producer-Single-Consumer

All threads

Sched-worker

High
contention

```
class NormalQueue:
```

```
    T* myHead;
```

```
    T* myTail;
```

```
NormalQueue::Push(T* aItem)
```

```
{
```

```
    if (myHead == nullptr)
```

```
        myHead = aItem;
```

```
    else
```

```
        myTail->myNext = aItem;
```

```
    myTail = aItem;
```

```
}
```

```
NormalQueue::Pop()
```

```
{
```

```
    if (myHead == nullptr)
```

```
        return nullptr;
```

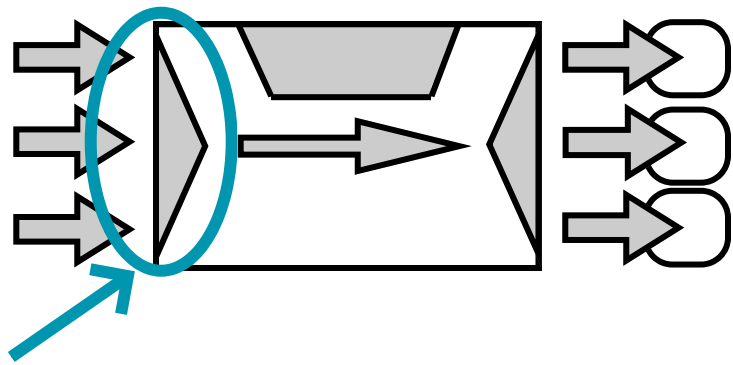
```
    T* res = myHead;
```

```
    myHead = res->myNext;
```

```
    return res;
```

```
}
```

Doing it in a lock-free
way is hardly possible
- **too many variables**



Front queue

Multi-Producer-Single-Consumer

■ — All threads — ■ ■ — Sched-worker — ■

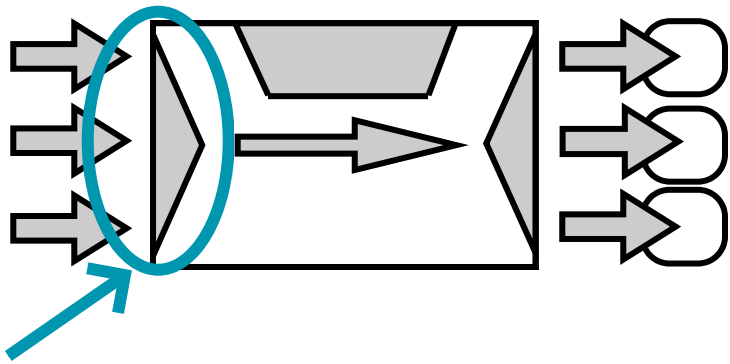
High
contention

```
class MPSCQueue:
    T* myTop;
```

```
MPSCQueue::Push(T* aItem)
{
    T* oldTop;
    do {
        oldTop = AtomicLoad(myTop);
        aItem->myNext = oldTop;
    } while (not AtomicCompareExchange(
        myTop, aItem, oldTop));
}

MPSCQueue::PopAll(T* aItem)
{
    T* top = AtomicExchange(myTop, nullptr);
    return ReverseList(top);
}
```

Make it a **stack** to
reduce the number of
variables



Front queue

Multi-Producer-Single-Consumer

All threads

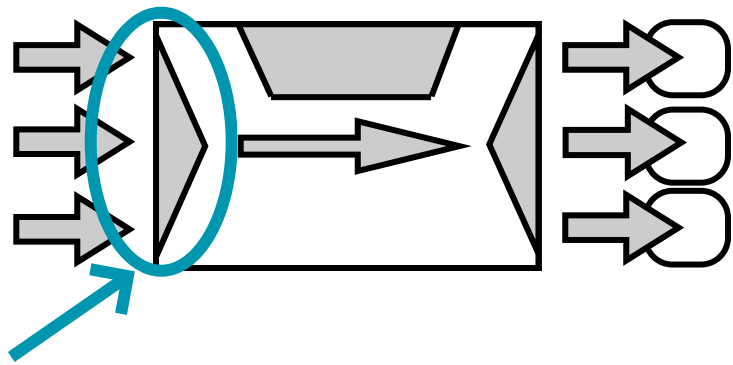
Sched-worker

High
contention

```
class MPSCQueue:  
    T* myTop;
```

```
MPSCQueue::Push(T* aItem)  
{  
    T* oldTop;  
    do {  
        oldTop = AtomicLoad(myTop);  
        aItem->myNext = oldTop;  
    } while (not AtomicCompareExchange(  
        myTop, aItem, oldTop));  
}  
  
MPSCQueue::PopAll(T* aItem)  
{  
    T* top = AtomicExchange(myTop, nullptr);  
    return ReverseList(top);  
}
```

Retry **atomic push** of
a new top. Stack
grows



Front queue

Multi-Producer-Single-Consumer

All threads

Sched-worker

High
contention

```
class MPSCQueue:
```

```
    T* myTop;
```

```
MPSCQueue::Push(T* aItem)
```

```
{
```

```
    T* oldTop;
```

```
    do {
```

```
        oldTop = AtomicLoad(myTop);
```

```
        aItem->myNext = oldTop;
```

```
    } while (not AtomicCompareExchange(
```

```
        myTop, aItem, oldTop));
```

```
}
```

```
MPSCQueue::PopAll(T* aItem)
```

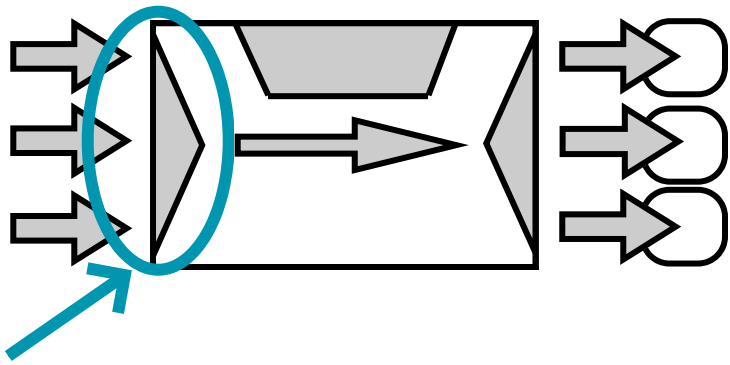
```
{
```

```
    T* top = AtomicExchange(myTop, nullptr);
```

```
    return ReverseList(top);
```

```
}
```

Pop **takes all** and
turns **stack to queue**



Front queue

Multi-Producer-Single-Consumer

■ — All threads — ■ ■ — Sched-worker — ■

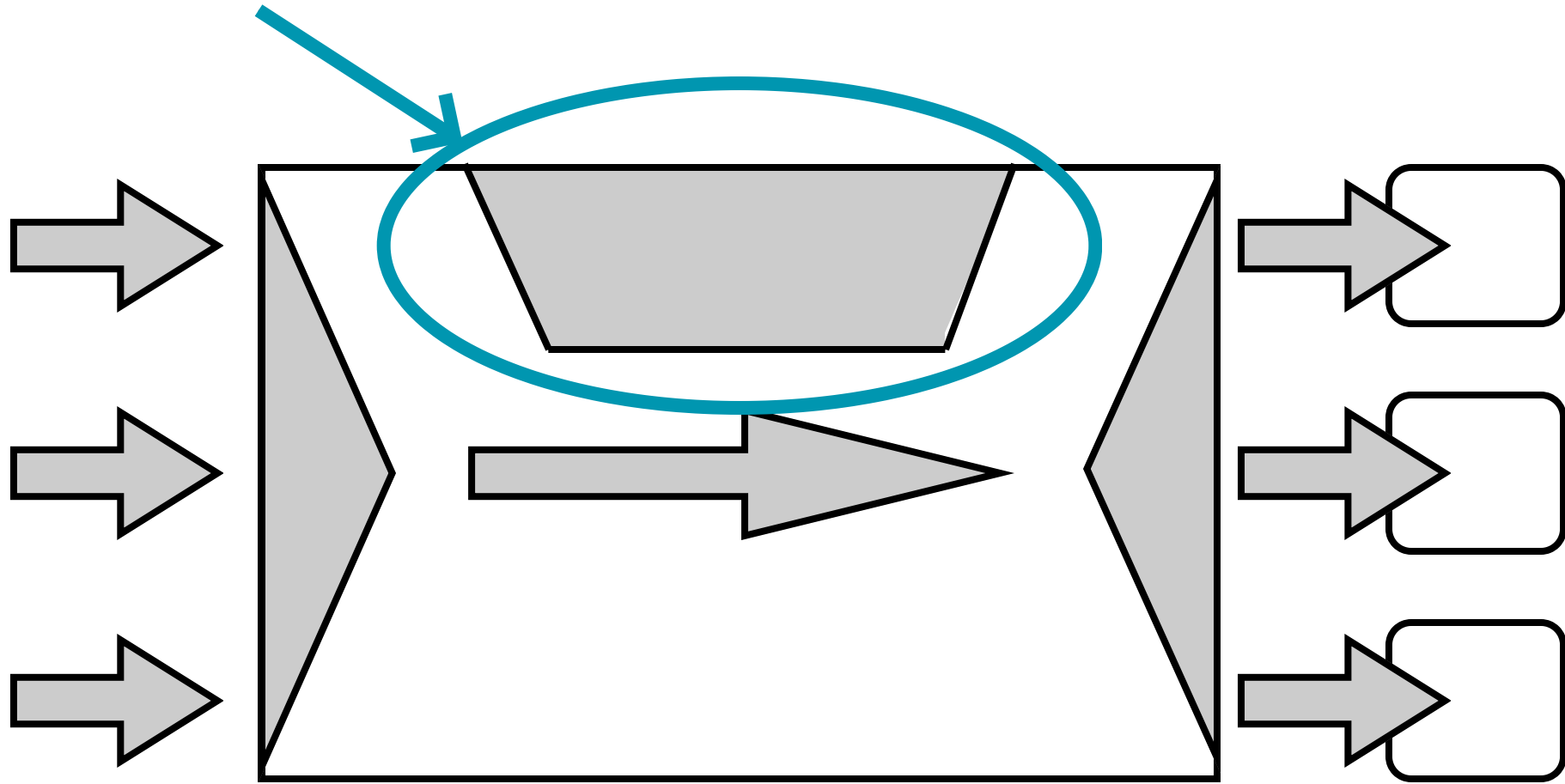
High
contention

Completely
lock-free

```
class MPSCQueue:  
    T* myTop;
```

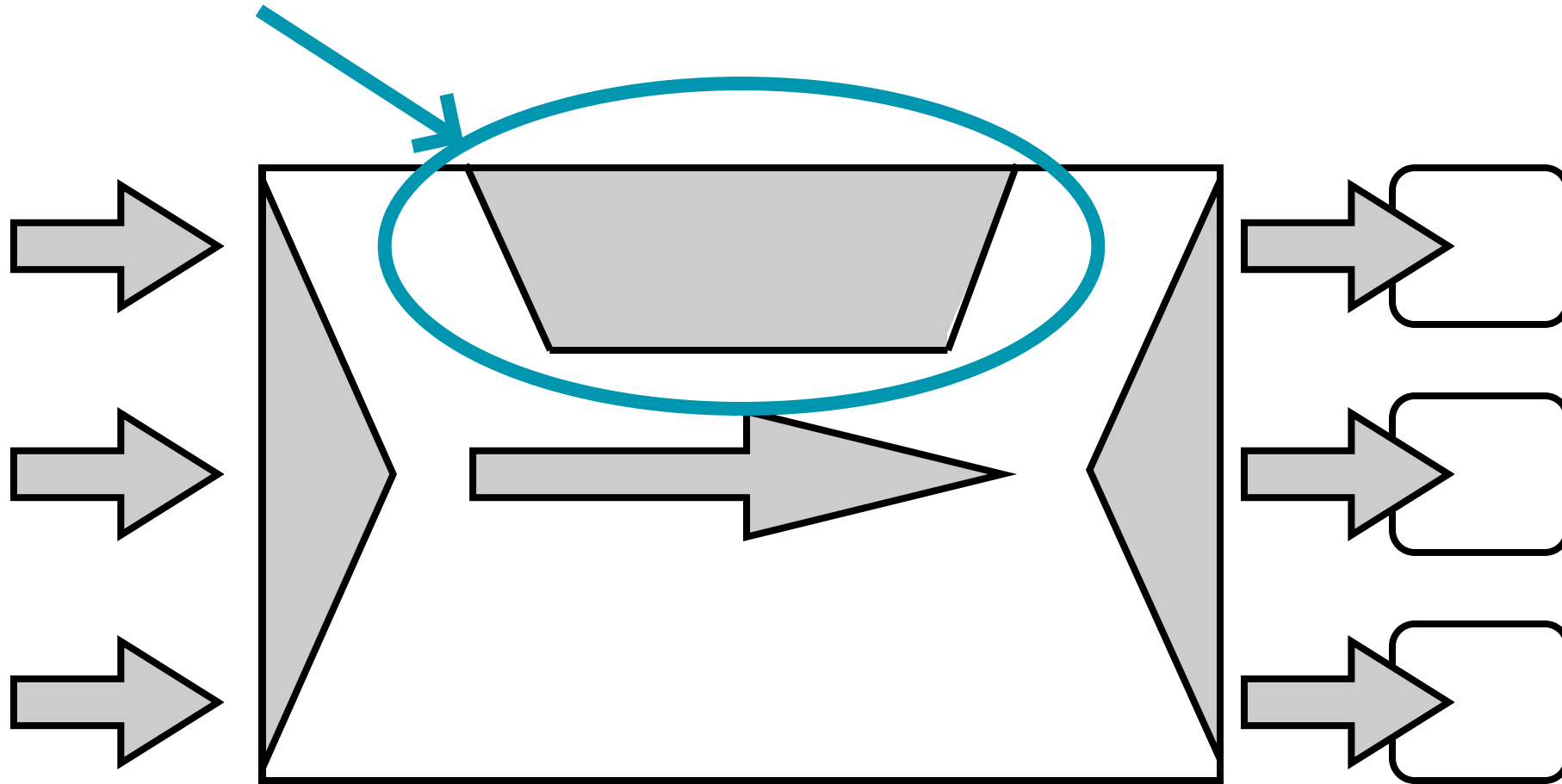
```
MPSCQueue::Push(T* aItem)  
{  
    T* oldTop;  
    do {  
        oldTop = AtomicLoad(myTop);  
        aItem->myNext = oldTop;  
    } while (not AtomicCompareExchange(  
        myTop, aItem, oldTop));  
}  
  
MPSCQueue::PopAll(T* aItem)  
{  
    T* top = AtomicExchange(myTop, nullptr);  
    return ReverseList(top);  
}
```

Wait queue



Wait queue

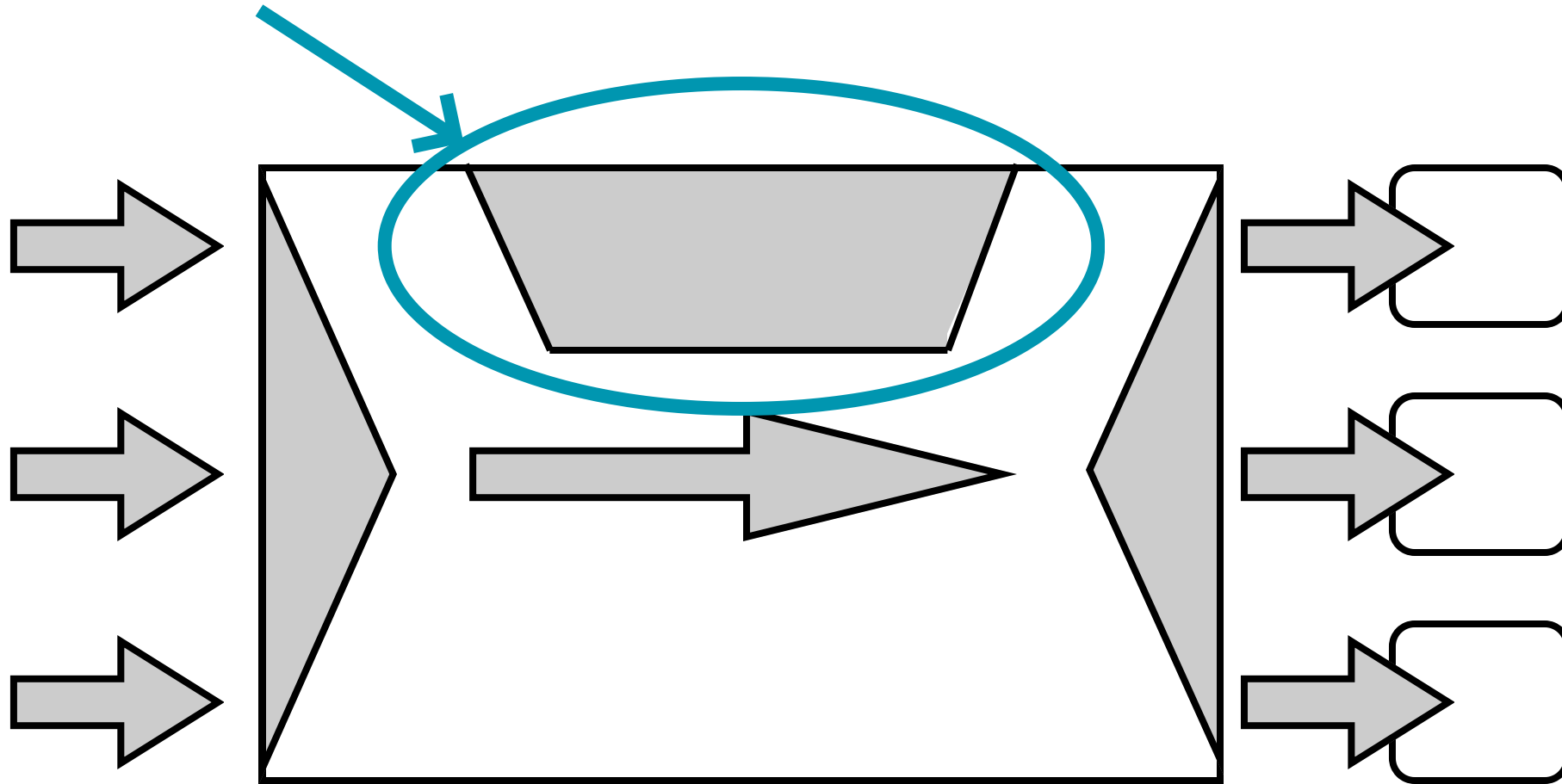
Quickly get
expired tasks

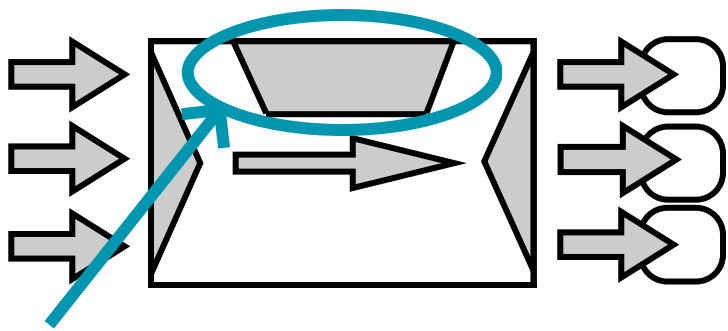


Wait queue

Binary heap
Sched-worker

Quickly get
expired tasks





Sort tasks by
deadlines - the
closest on top

Wait queue

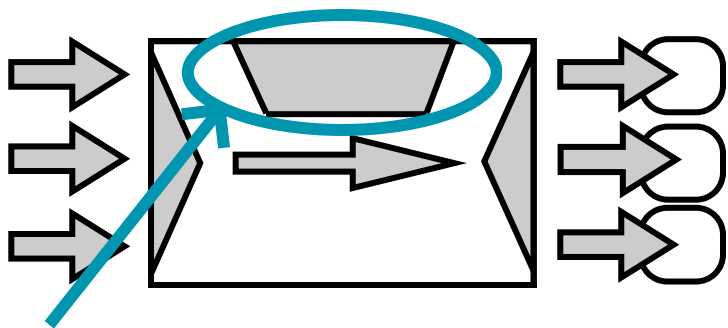
Binary heap
—
Sched-worker

$O(\log(N))$ – *update*

$O(1)$ – *get top*

Quickly get
expired tasks

Very good
time
complexity



Sort tasks by
deadlines - the
closest on top

Perfectly balanced
binary tree

Node \geq children

Wait queue

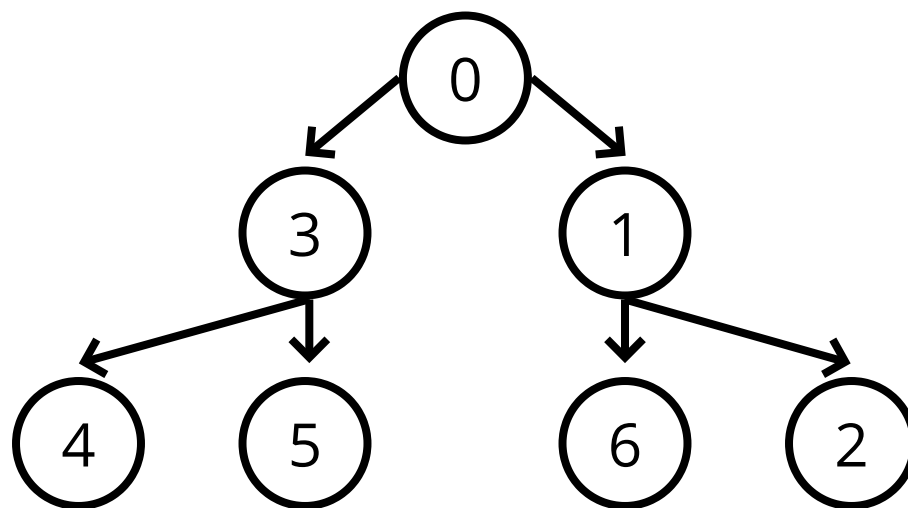
Binary heap
—
Sched-worker

$O(\log(N))$ – *update*

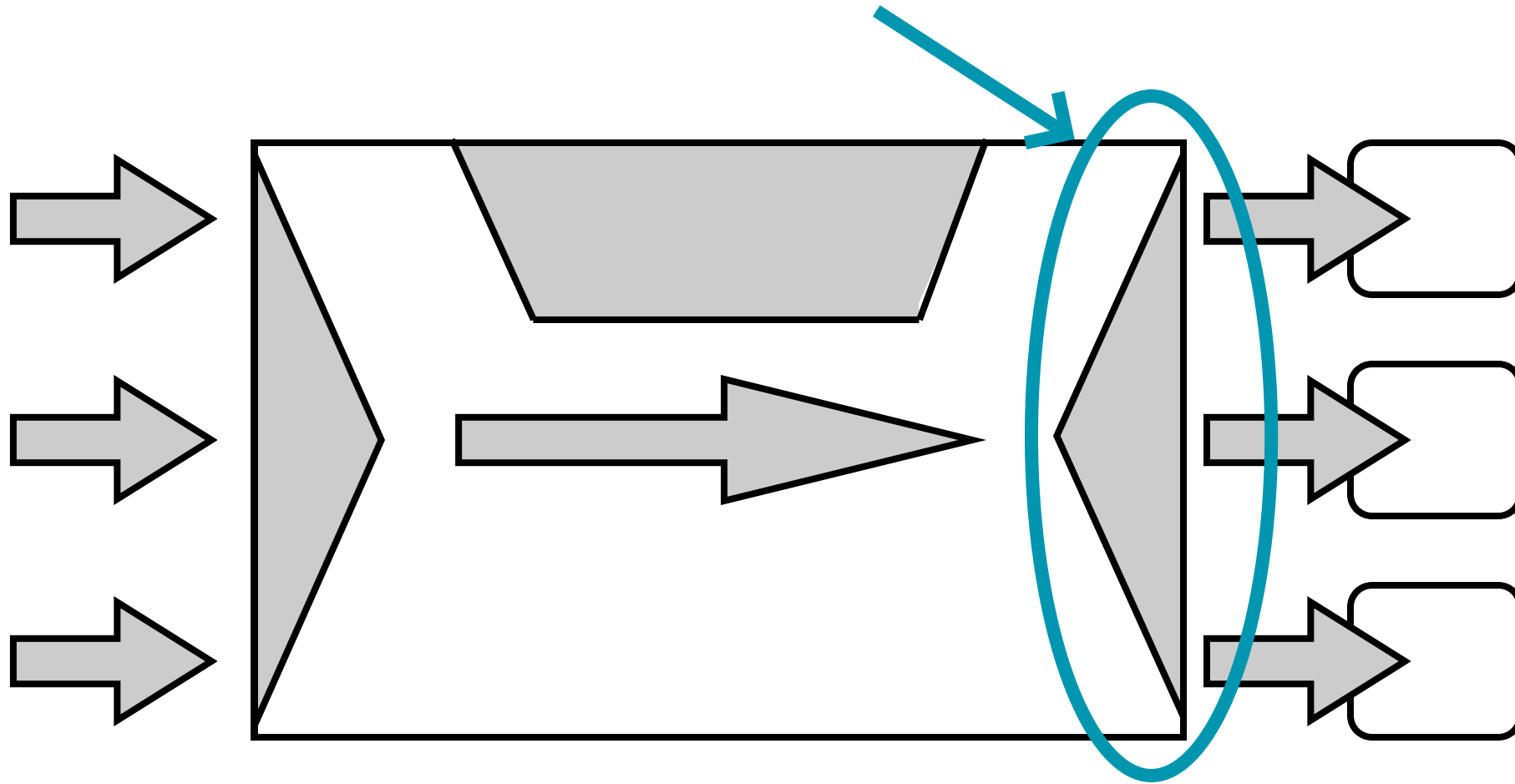
$O(1)$ – *get top*

Quickly get
expired tasks

Very good
time
complexity



Ready queue

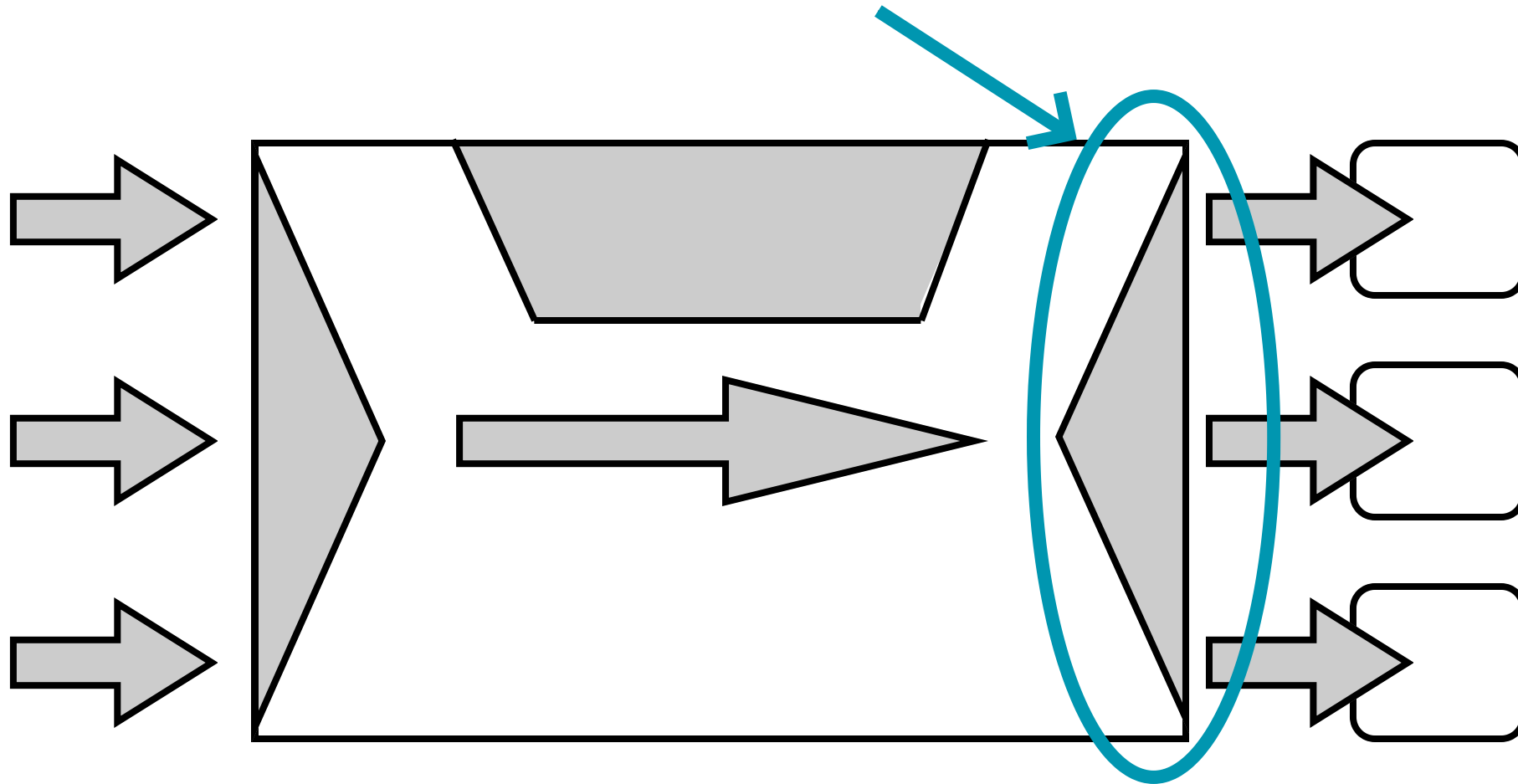


Ready queue

Multi-Consumer-Single-Producer

■ ——— ■
All threads

■ ——— ■
Sched-worker

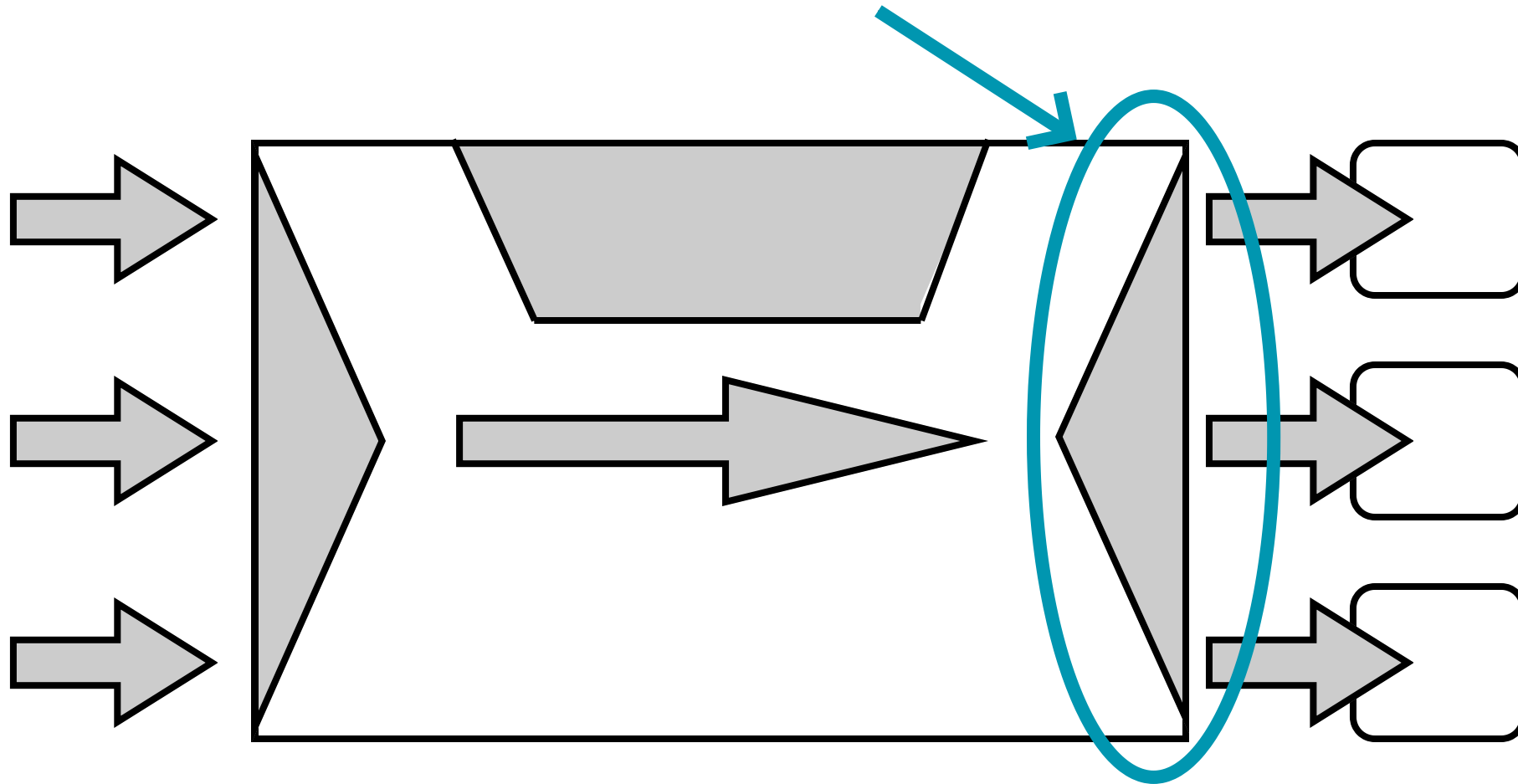


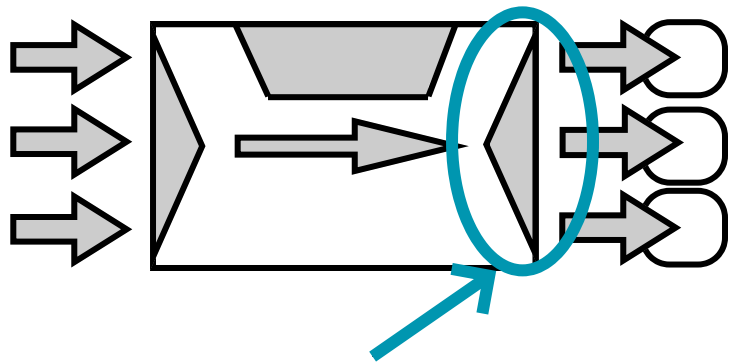
Ready queue

Multi-Consumer-Single-Producer

■ — All threads — ■ — Sched-worker — ■

High
contention





Ready queue

Multi-Consumer-Single-Producer

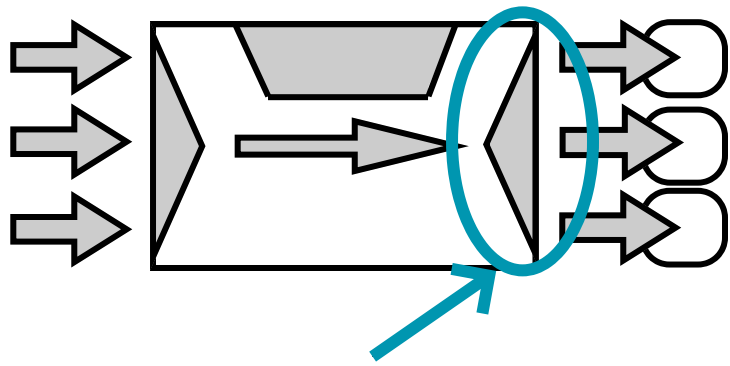
All threads

Sched-worker

High
contention

There is **no** simple unbounded lock-free MCSP queue

* Google "**ABA-problem**" about why



Ready queue

Multi-Consumer-Single-Producer

All threads

Sched-worker

High
contention

There is **no** simple unbounded lock-free MCSP queue

* Google "**ABA-problem**" about why

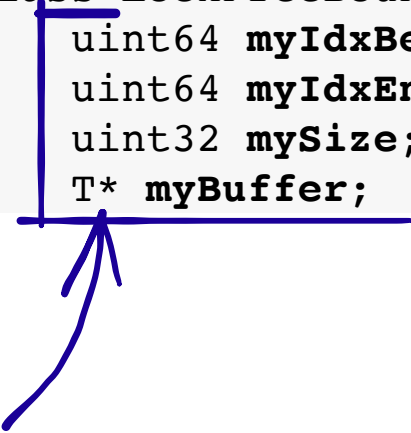
But there **are** these:

Unbounded lock-based
MCSP queue

Bounded lock-free MCSP
queue

Ready queue - Bounded Lock-Free

```
class LockFreeBounded:  
    uint64 myIdxBegin;  
    uint64 myIdxEnd;  
    uint32 mySize;  
    T* myBuffer;
```



Cyclic array with
atomic indexes

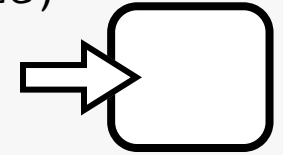
```
LockFreeBounded::Push(T* aItem)  
{  
    uint64 idxEnd = AtomicLoad(myIdxEnd);  
    if (idxEnd - AtomicLoad(myIdxBegin) == mySize)  
        return false;  
    myBuffer[idxEnd % mySize] = aItem;  
    AtomicExchange(myIdxEnd, idxEnd + 1);  
}
```

```
LockFreeBounded::Pop()  
{  
    T* res;  
    do {  
        uint64 idxBegin = AtomicLoad(myIdxBegin);  
        if (idxBegin == AtomicLoad(myIdxEnd))  
            return nullptr;  
        res = myBuffer[idxBegin % mySize];  
    } while (not AtomicCompareExchange(  
        myIdxBegin, idxBegin + 1, idxBegin));  
    return res;  
}
```

Ready queue - Bounded Lock-Free

```
class LockFreeBounded:  
    uint64 myIdxBegin;  
    uint64 myIdxEnd;  
    uint32 mySize;  
    T* myBuffer;
```

```
LockFreeBounded::Push(T* aItem)  
{  
    uint64 idxEnd = AtomicLoad(myIdxEnd);  
    if (idxEnd - AtomicLoad(myIdxBegin) == mySize)  
        return false;  
    myBuffer[idxEnd % mySize] = aItem;  
    AtomicExchange(myIdxEnd, idxEnd + 1);  
}
```



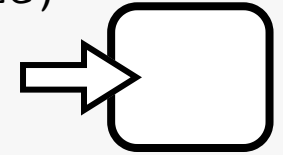
Single producer
atomically bumps
'end index'

```
LockFreeBounded::Pop()  
{  
    T* res;  
    do {  
        uint64 idxBegin = AtomicLoad(myIdxBegin);  
        if (idxBegin == AtomicLoad(myIdxEnd))  
            return nullptr;  
        res = myBuffer[idxBegin % mySize];  
    } while (not AtomicCompareExchange(  
        myIdxBegin, idxBegin + 1, idxBegin));  
    return res;  
}
```

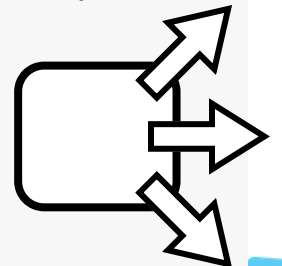
Ready queue - Bounded Lock-Free

```
class LockFreeBounded:  
    uint64 myIdxBegin;  
    uint64 myIdxEnd;  
    uint32 mySize;  
    T* myBuffer;
```

```
LockFreeBounded::Push(T* aItem)  
{  
    uint64 idxEnd = AtomicLoad(myIdxEnd);  
    if (idxEnd - AtomicLoad(myIdxBegin) == mySize)  
        return false;  
    myBuffer[idxEnd % mySize] = aItem;  
    AtomicExchange(myIdxEnd, idxEnd + 1);  
}
```



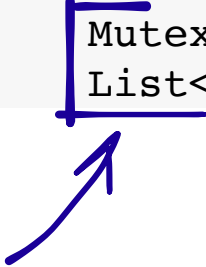
```
LockFreeBounded::Pop()  
{  
    T* res;  
    do {  
        uint64 idxBegin = AtomicLoad(myIdxBegin);  
        if (idxBegin == AtomicLoad(myIdxEnd))  
            return nullptr;  
        res = myBuffer[idxBegin % mySize];  
    } while (not AtomicCompareExchange(  
        myIdxBegin, idxBegin + 1, idxBegin));  
    return res;  
}
```



Consumers read by
atomically incremented
'begin index'

Ready queue - Unbounded Locked

```
class LockedUnbounded:  
    Mutex myLock;  
    List<T> myQueue;
```



Trivial mutex and list

```
LockedUnbounded::Push(T* aItem)  
{  
    myLock.Lock();  
    myQueue.Append(aItem);  
    myLock.Unlock();  
}  
  
LockedUnbounded::Pop()  
{  
    myLock.Lock();  
    T* res = nullptr;  
    if (myQueue.IsEmpty())  
        res = myQueue.PopFirst();  
    myLock.Unlock();  
    return res;  
}
```

Ready queue - Unbounded Locked

```
class LockedUnbounded:  
    Mutex myLock;  
    List<T> myQueue;
```

Lock on push and pop

```
LockedUnbounded::Push(T* aItem)
```

```
{  
    myLock.Lock();  
    myQueue.Append(aItem);  
    myLock.Unlock();  
}
```

```
LockedUnbounded::Pop()
```

```
{  
    myLock.Lock();  
    T* res = nullptr;  
    if (myQueue.IsEmpty())  
        res = myQueue.PopFirst();  
    myLock.Unlock();  
    return res;  
}
```

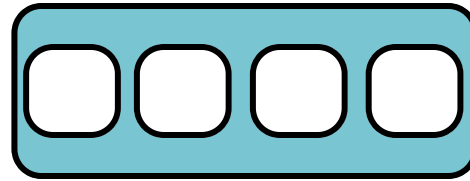
Ready queue

Ready queue



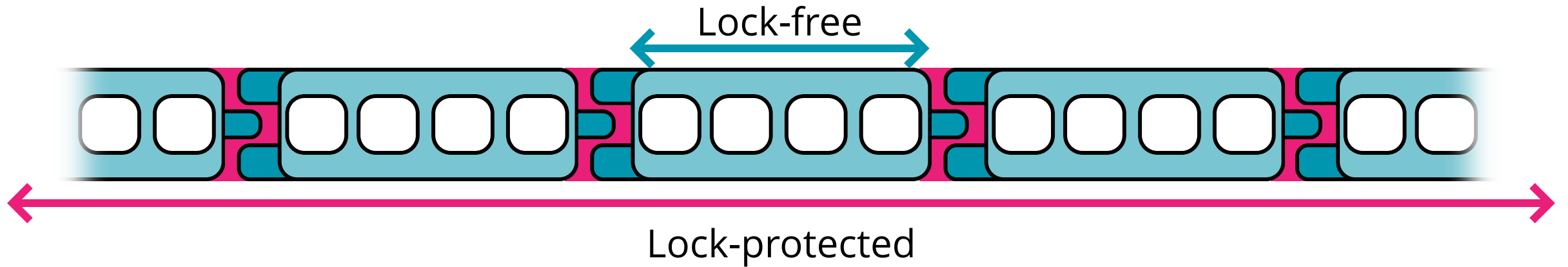
Ready queue

Lock-free



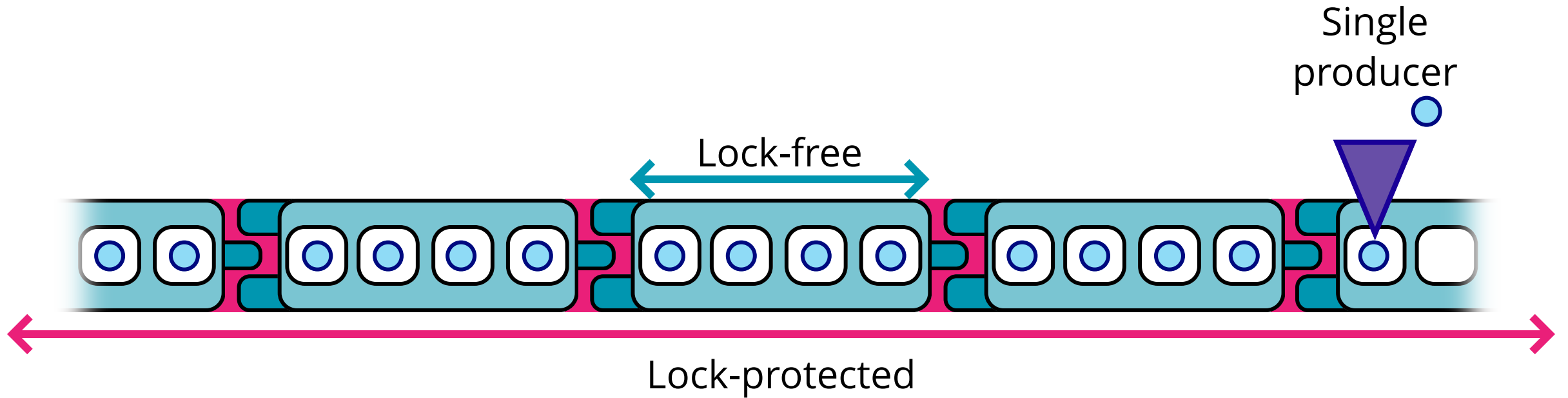
Ready queue

Lock-based **queue** of lock-free **queues**



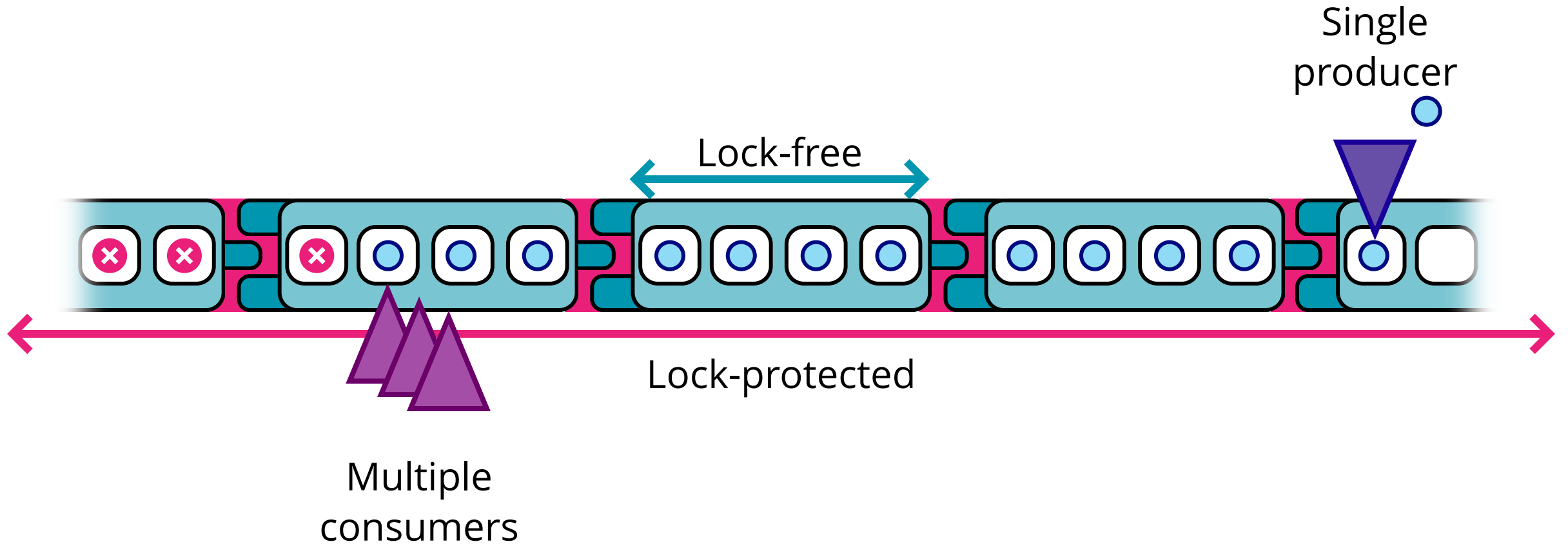
Ready queue

Lock-based **queue** of lock-free **queues**



Ready queue

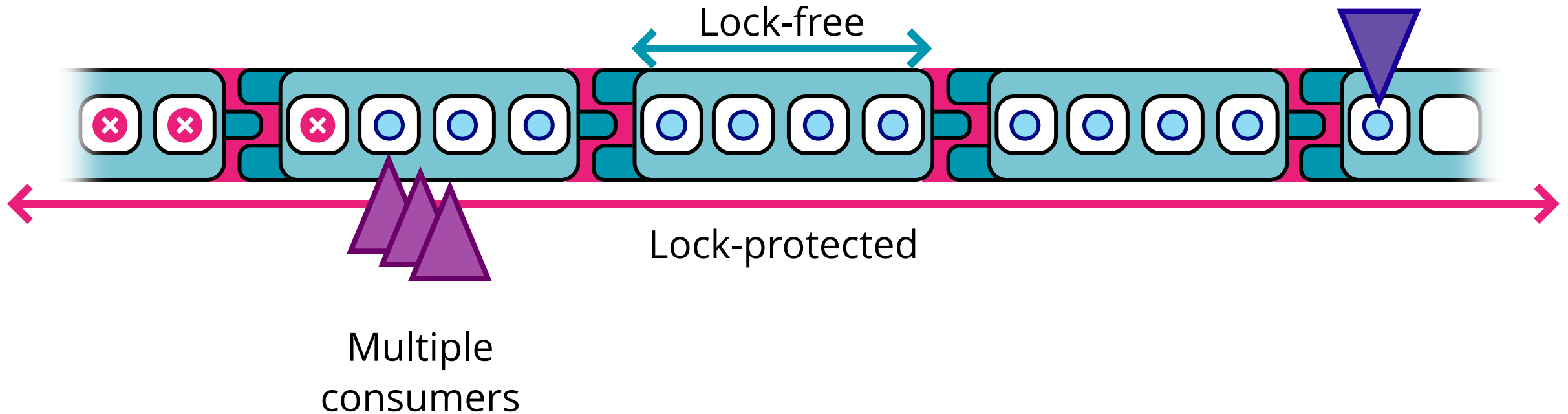
Lock-based **queue** of lock-free **queues**



Ready queue

Lock-based queue of lock-free queues

Lock is taken once per
sub-queue size



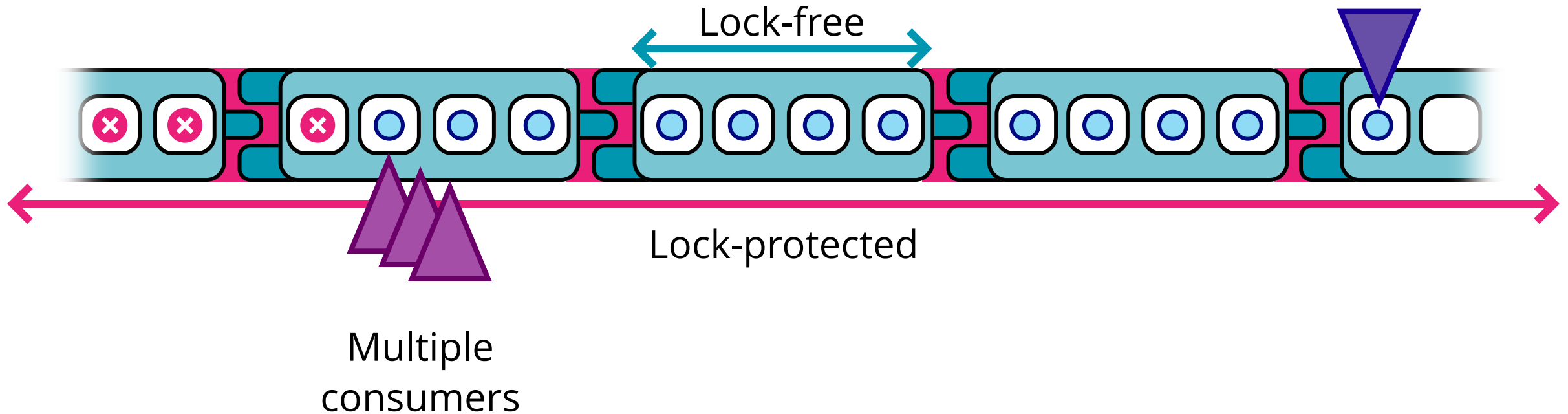
Ready queue

Lock-based **queue of** lock-free **queues**

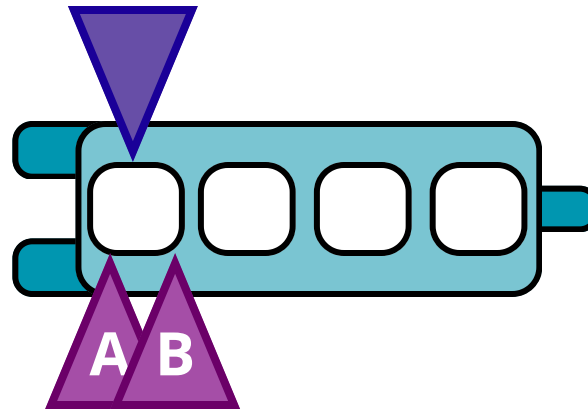
Lock is taken once per
sub-queue size

Consumers need an
explicit state

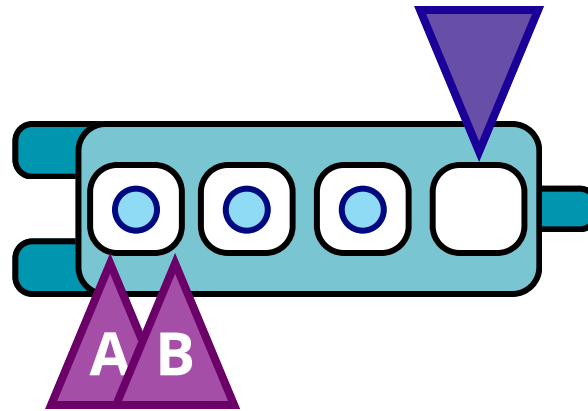
Single
producer



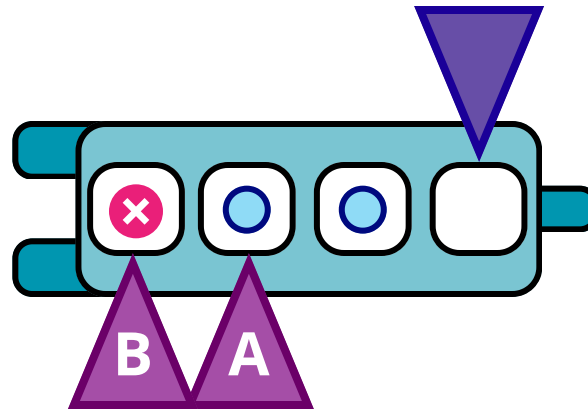
Ready queue example



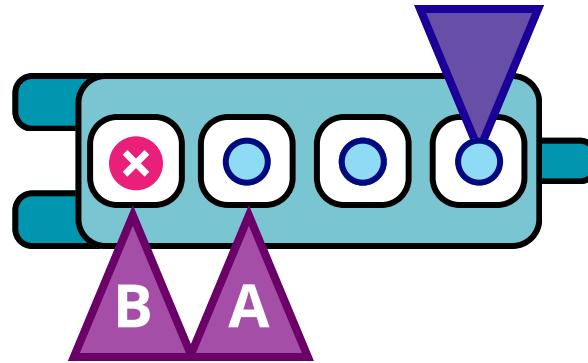
Ready queue example



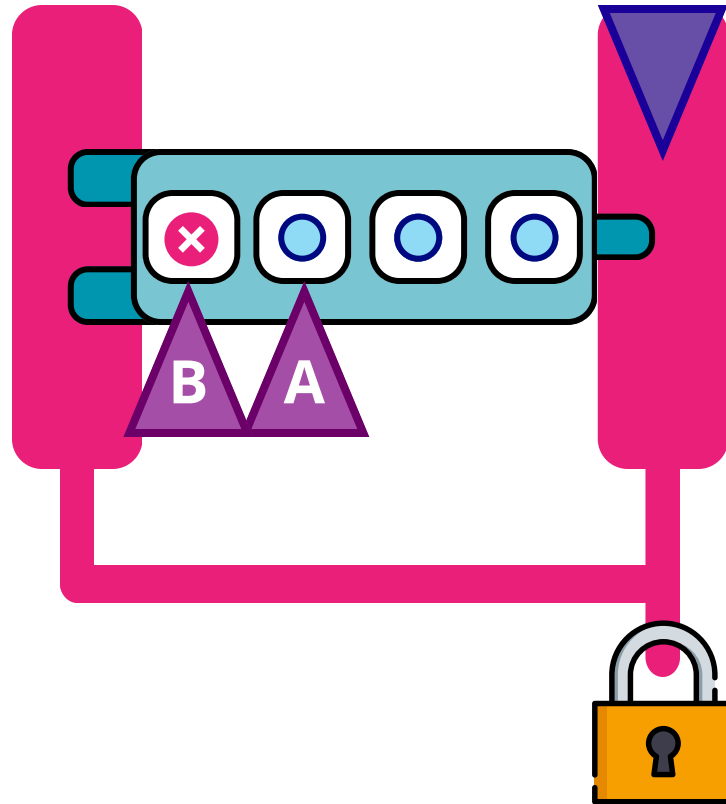
Ready queue example



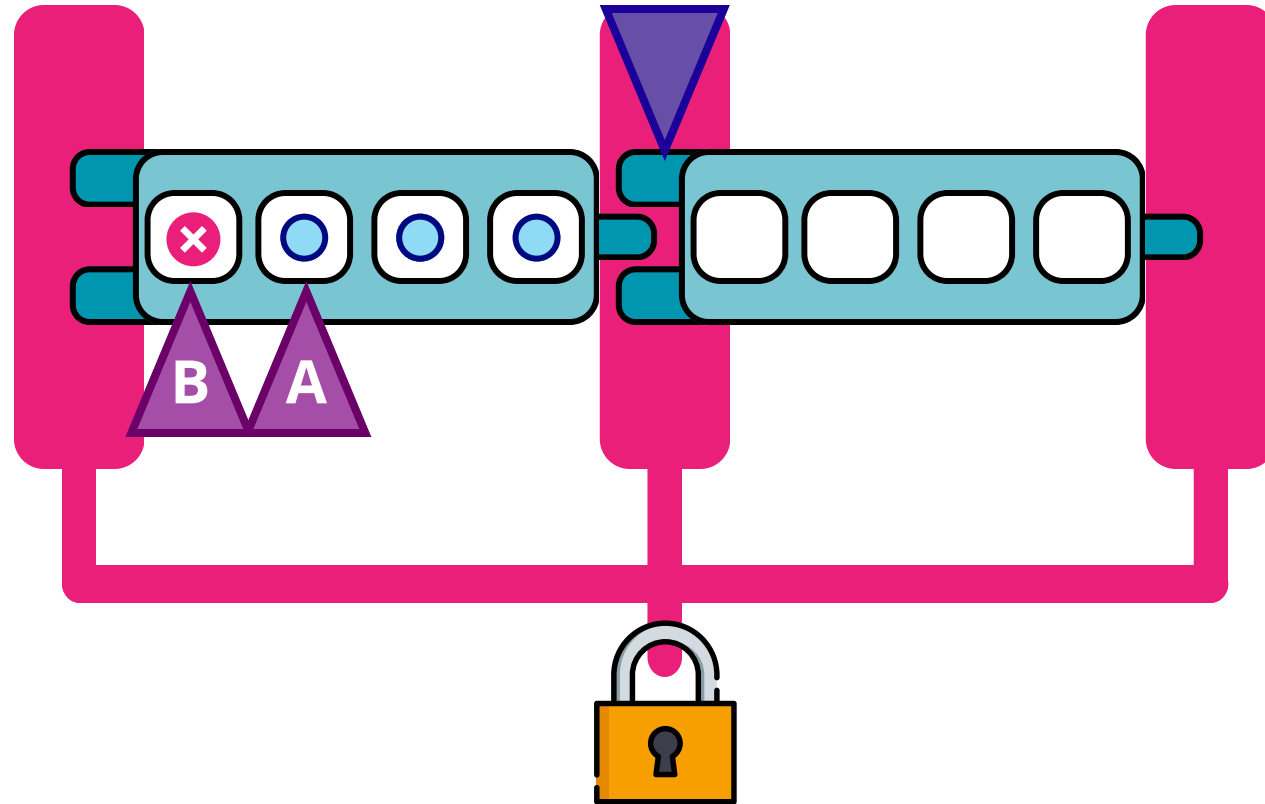
Ready queue example



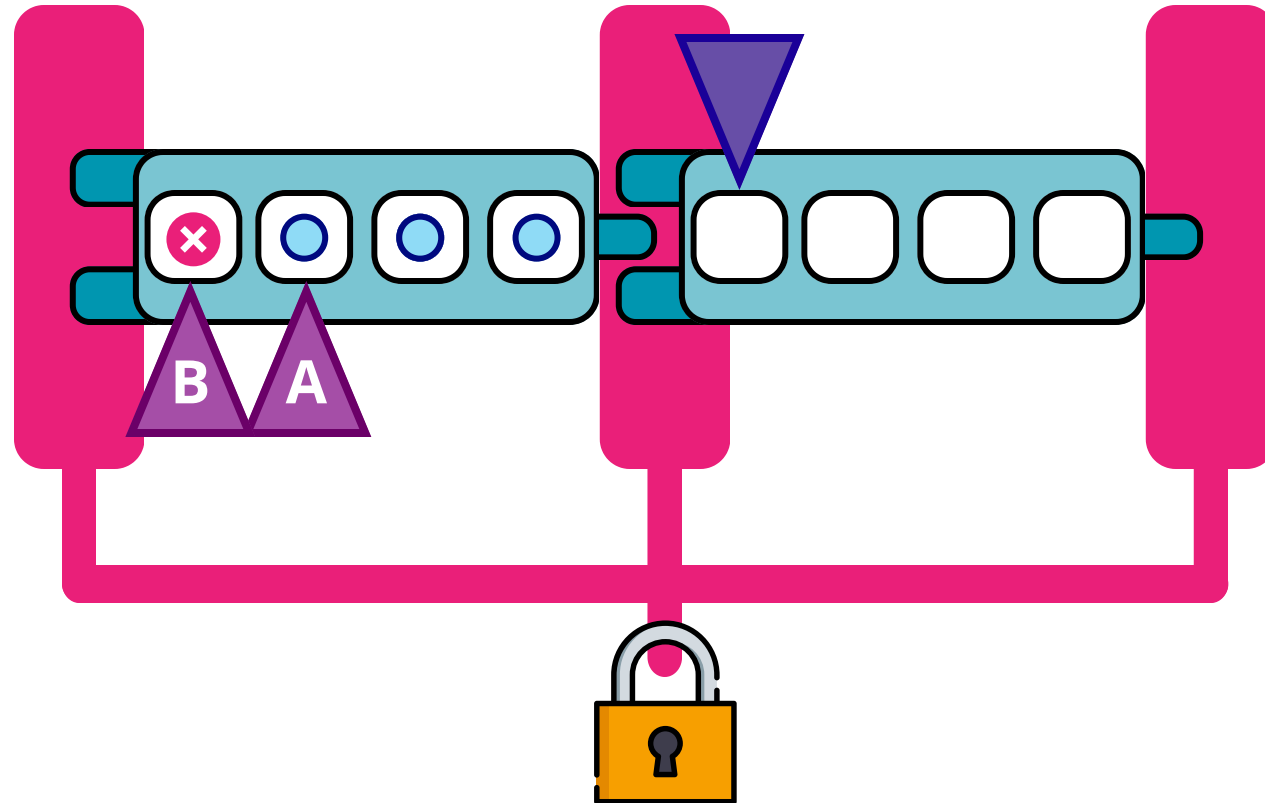
Ready queue example



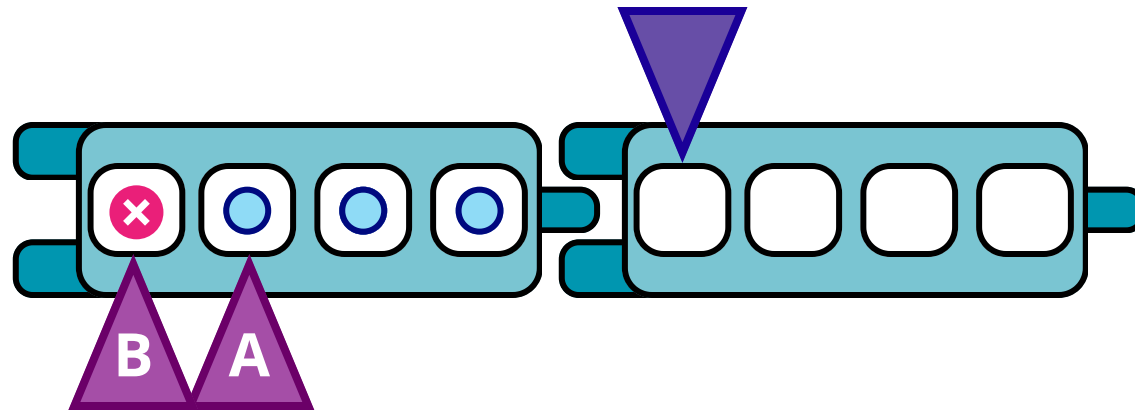
Ready queue example



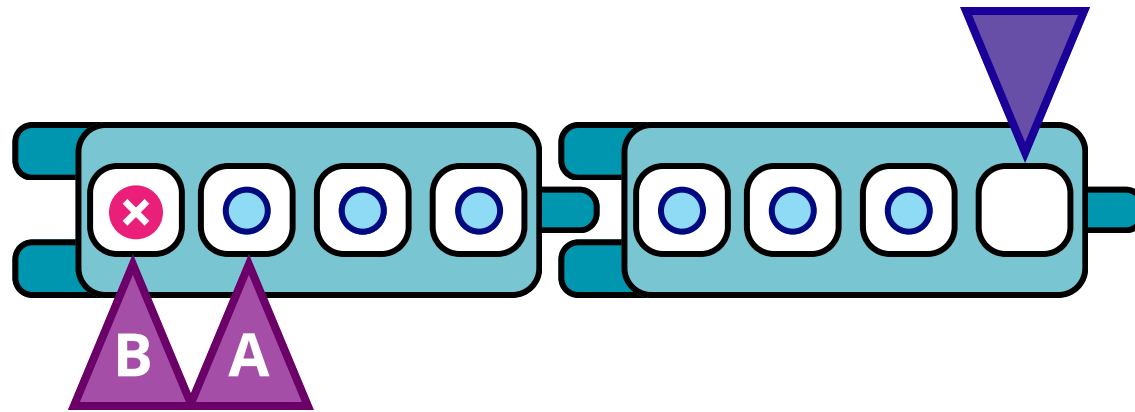
Ready queue example



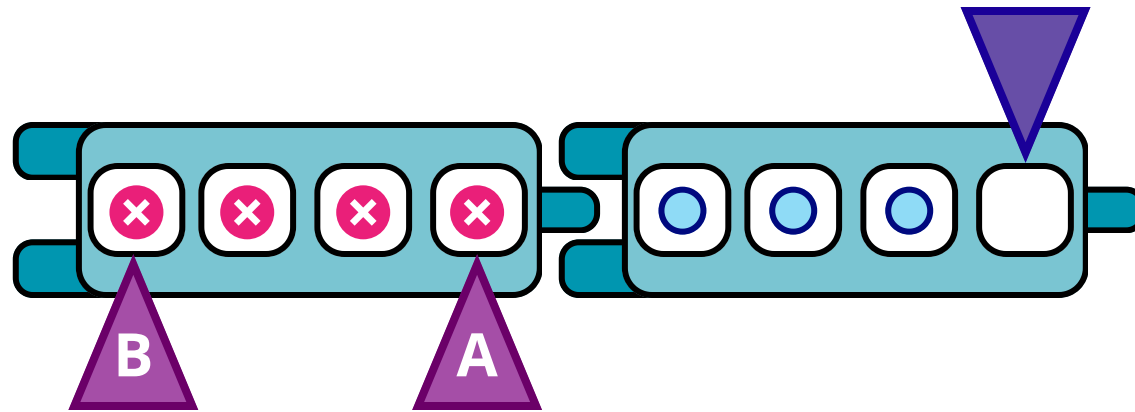
Ready queue example



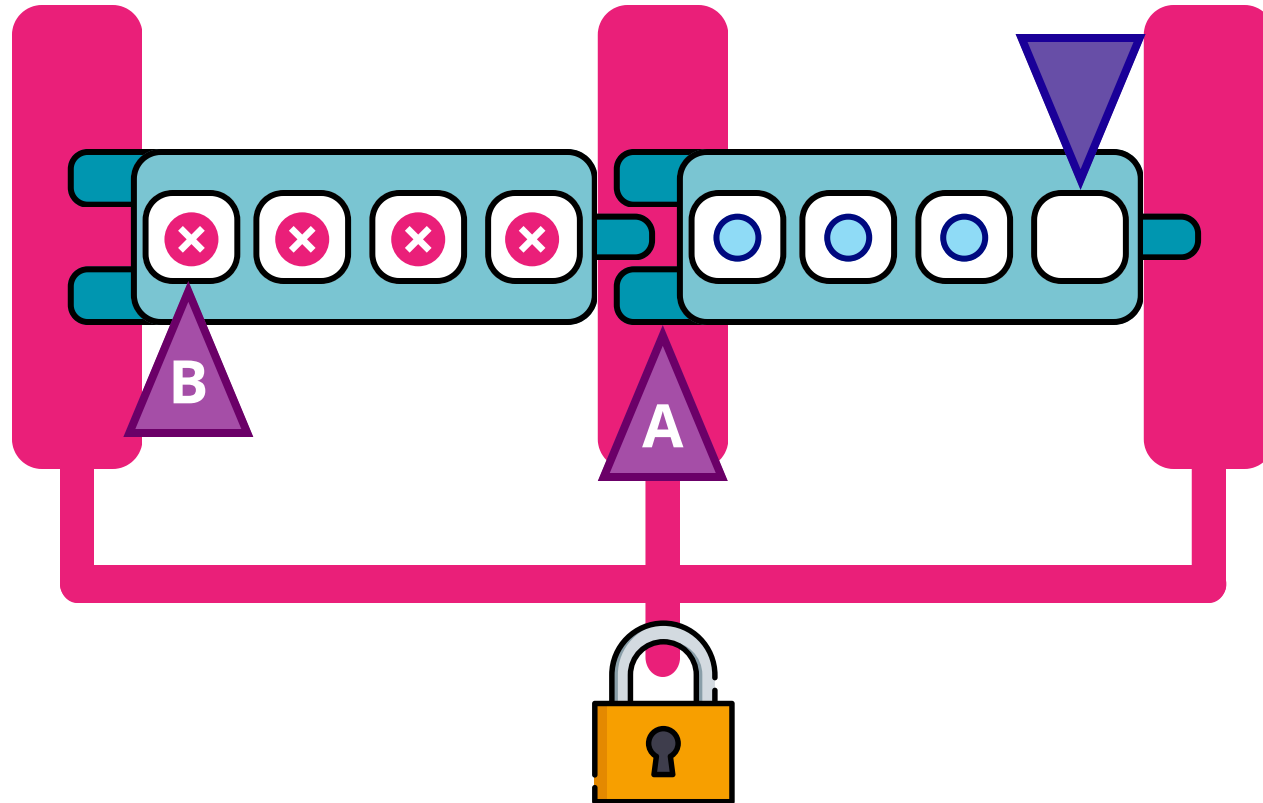
Ready queue example



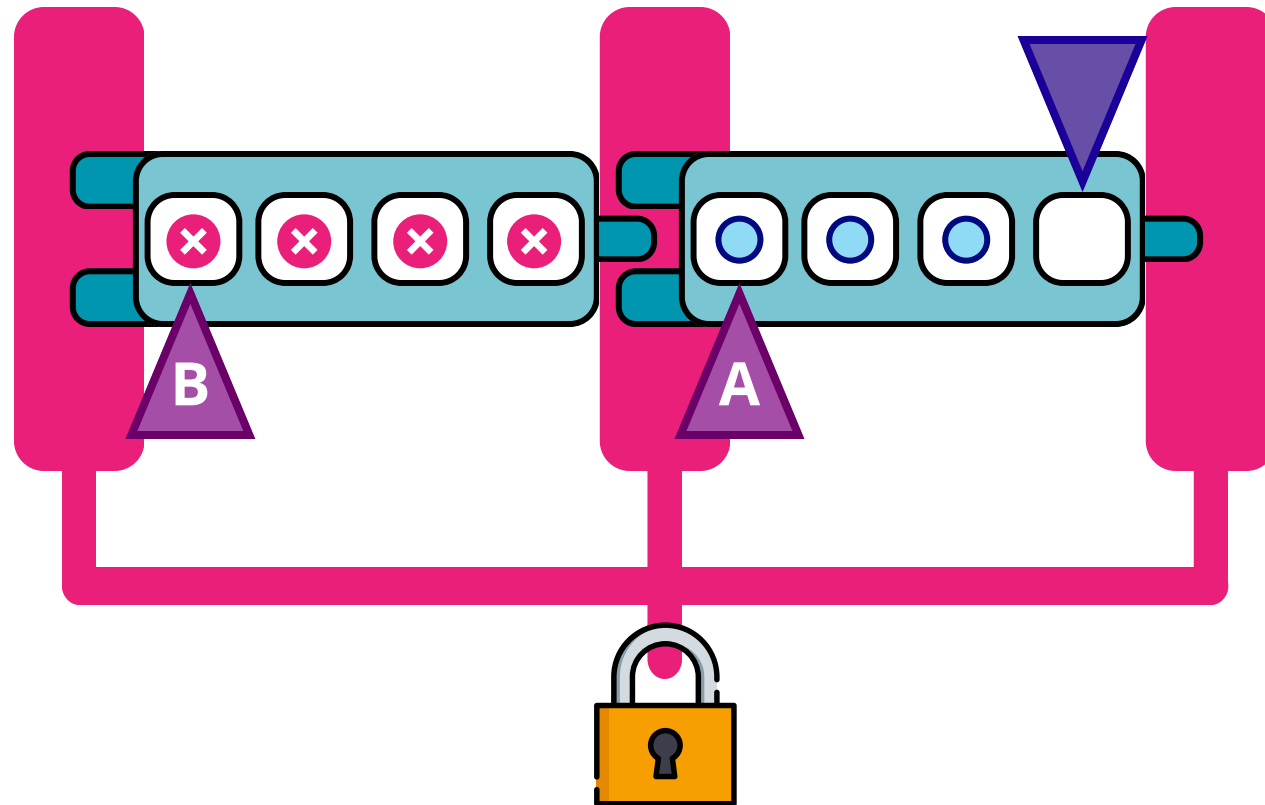
Ready queue example



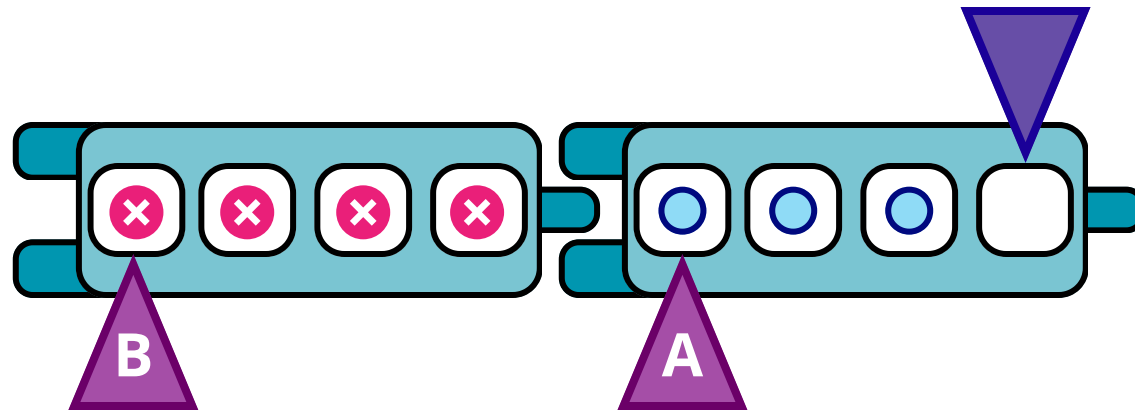
Ready queue example



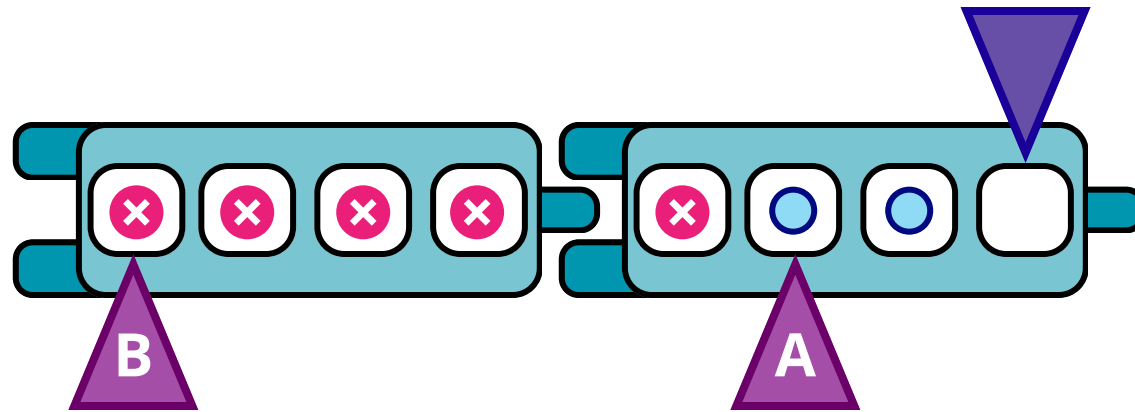
Ready queue example



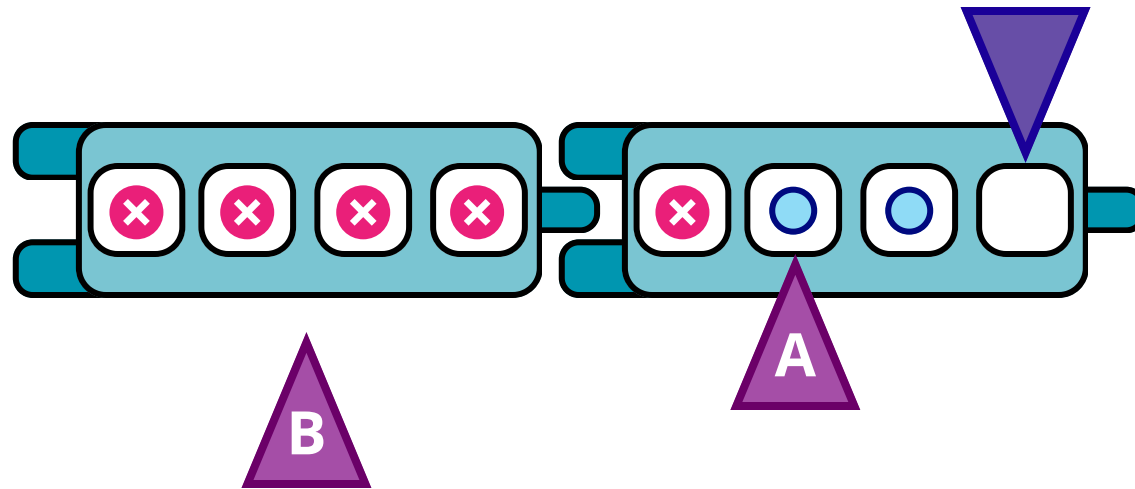
Ready queue example



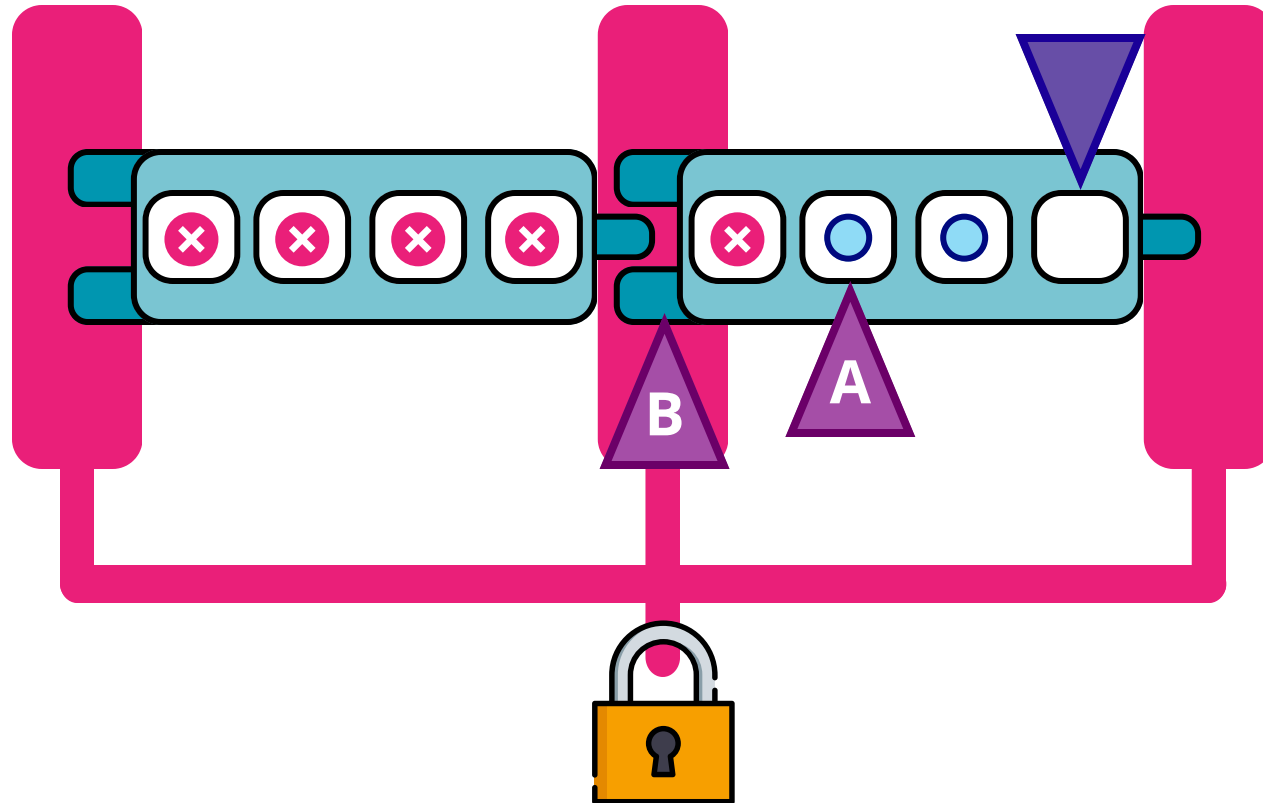
Ready queue example



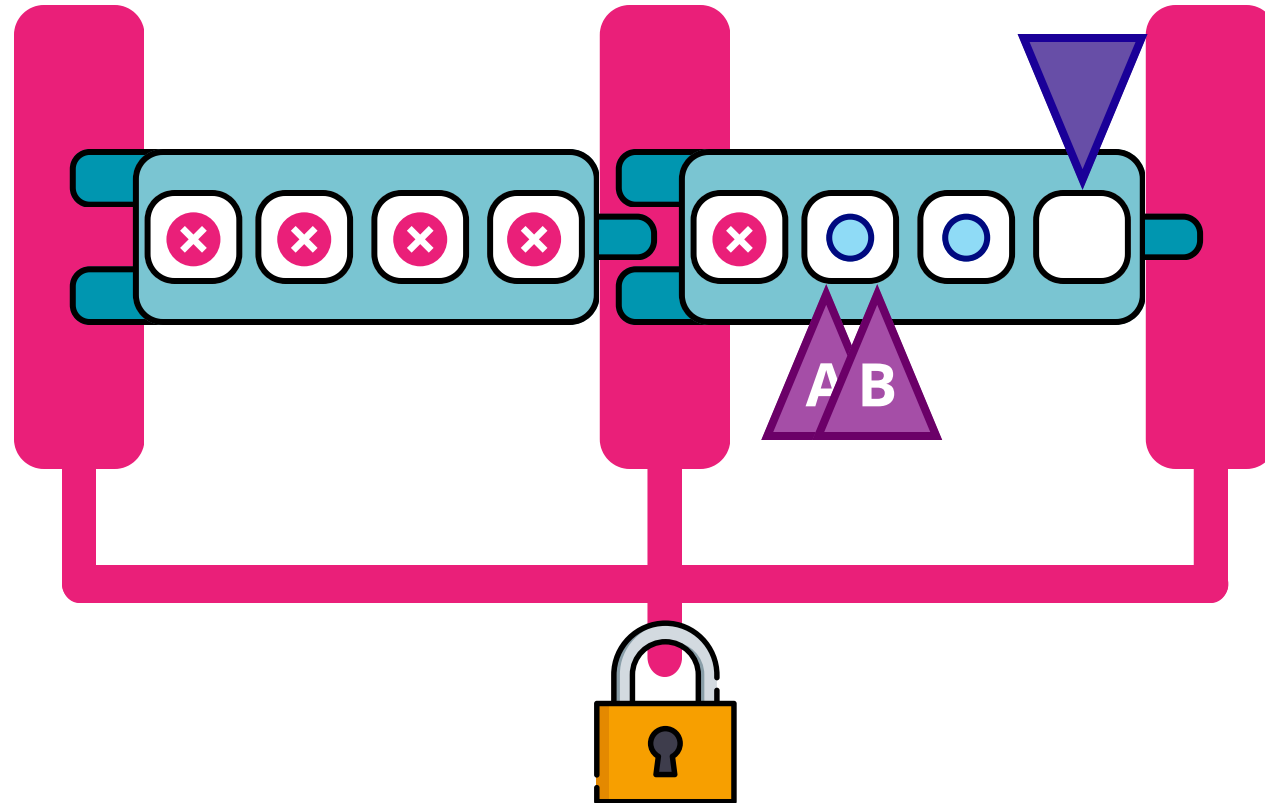
Ready queue example



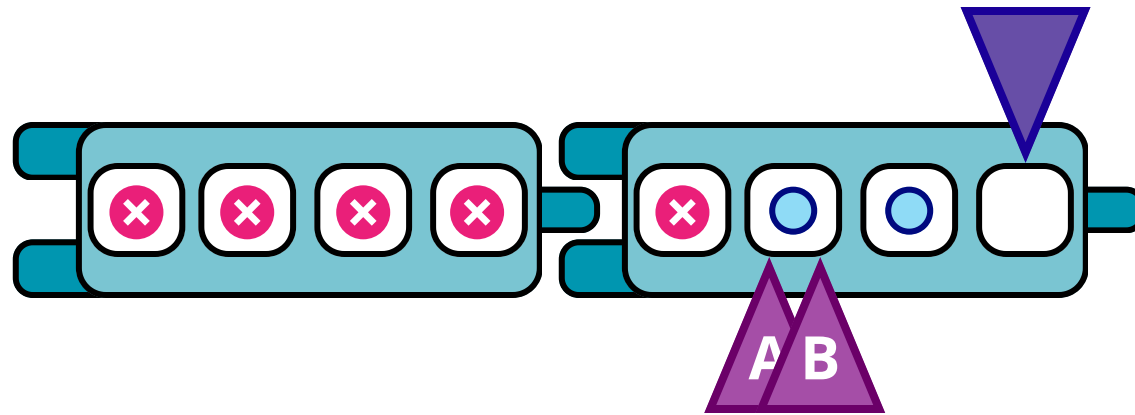
Ready queue example



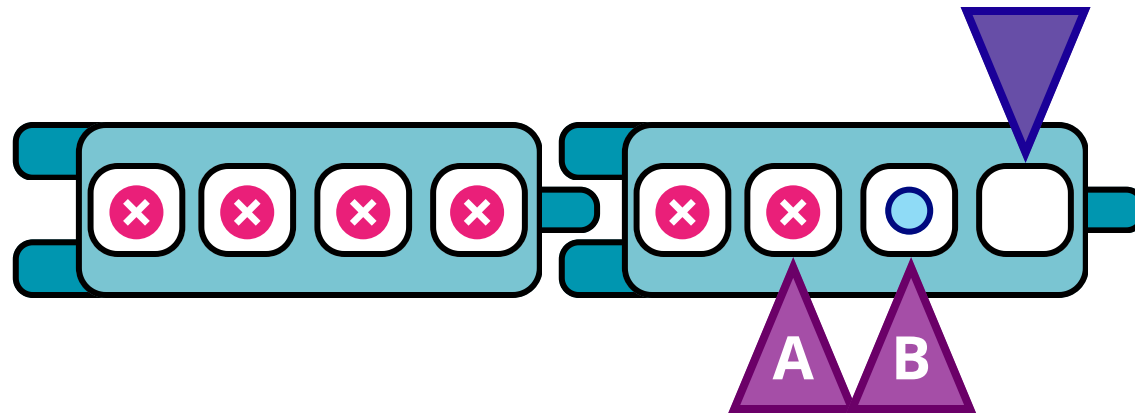
Ready queue example



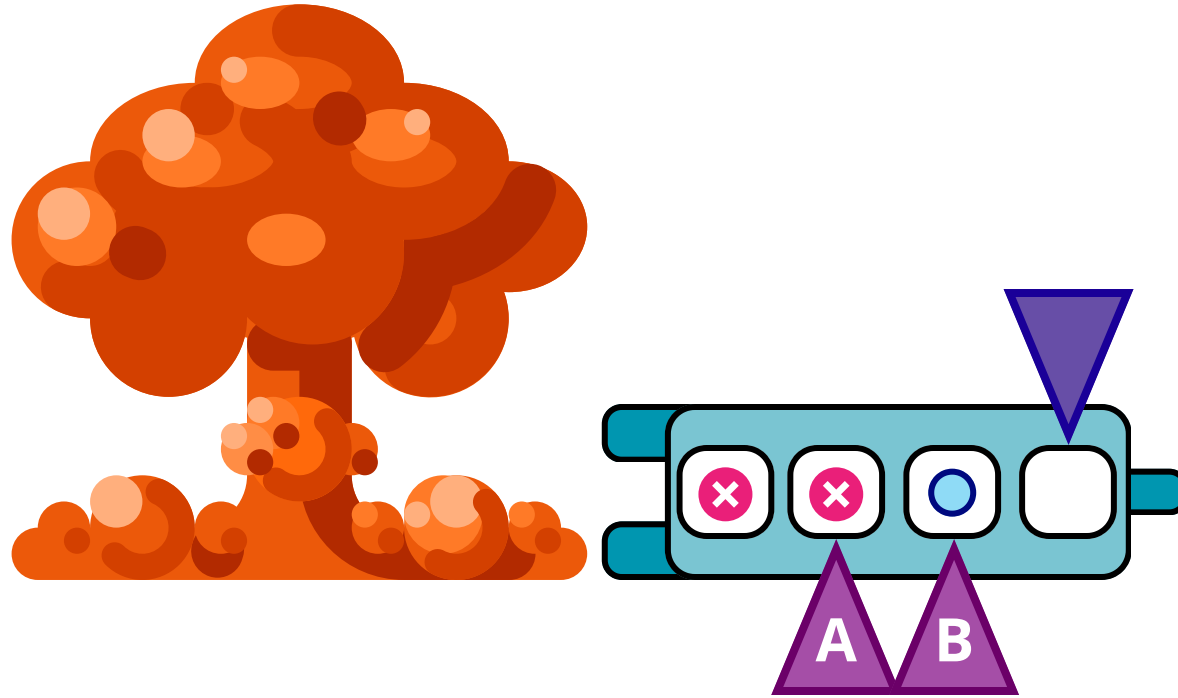
Ready queue example



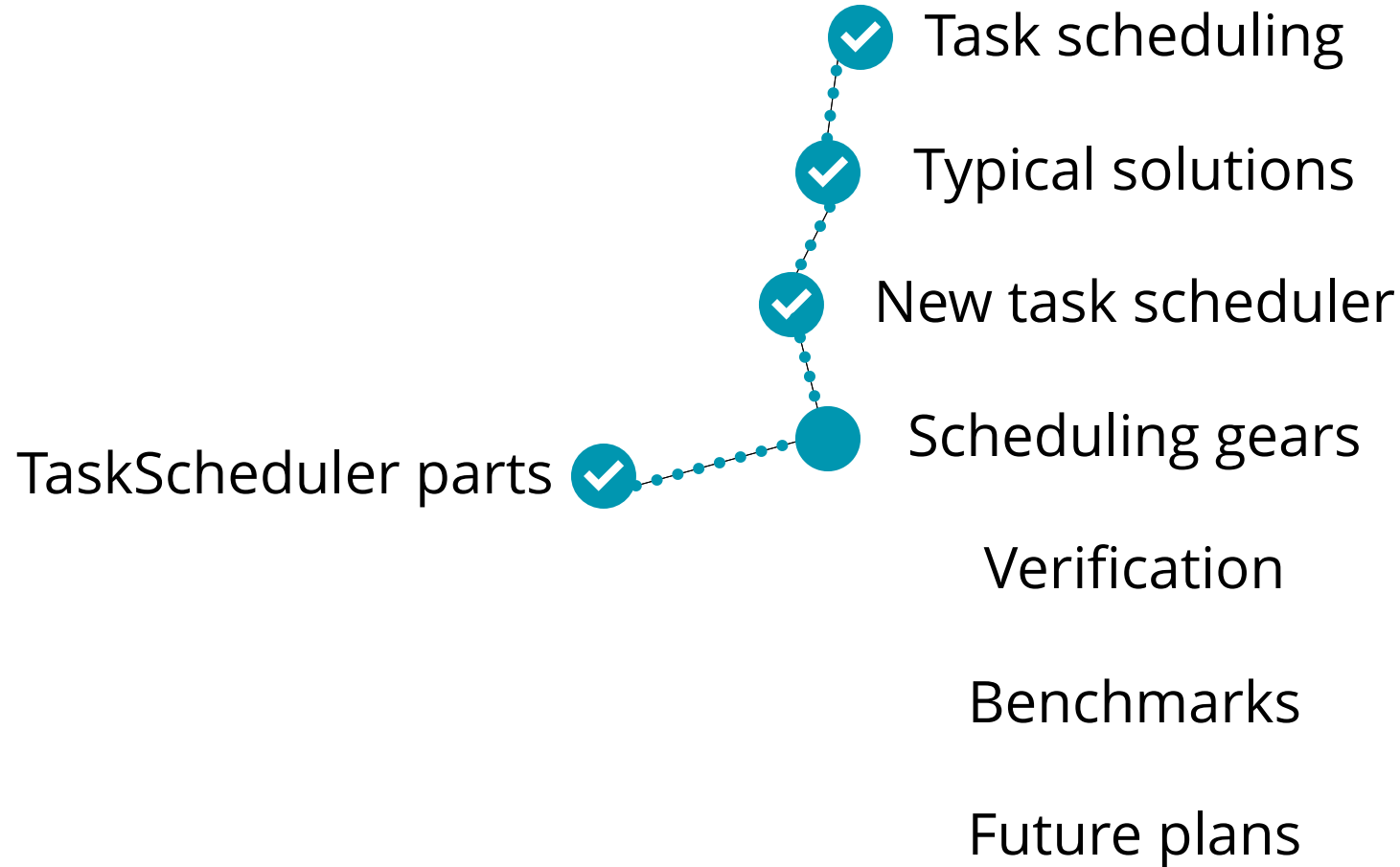
Ready queue example



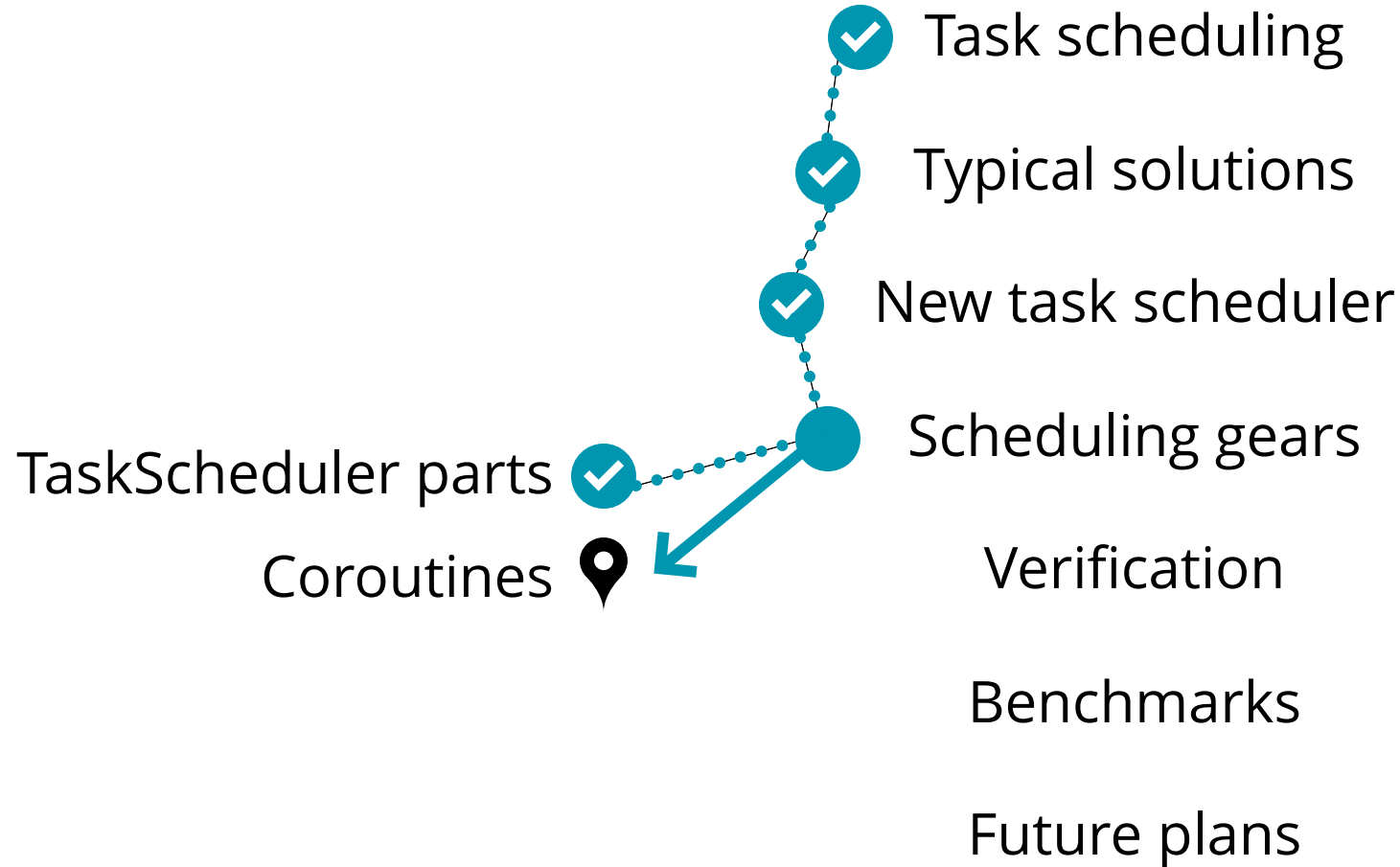
Ready queue example



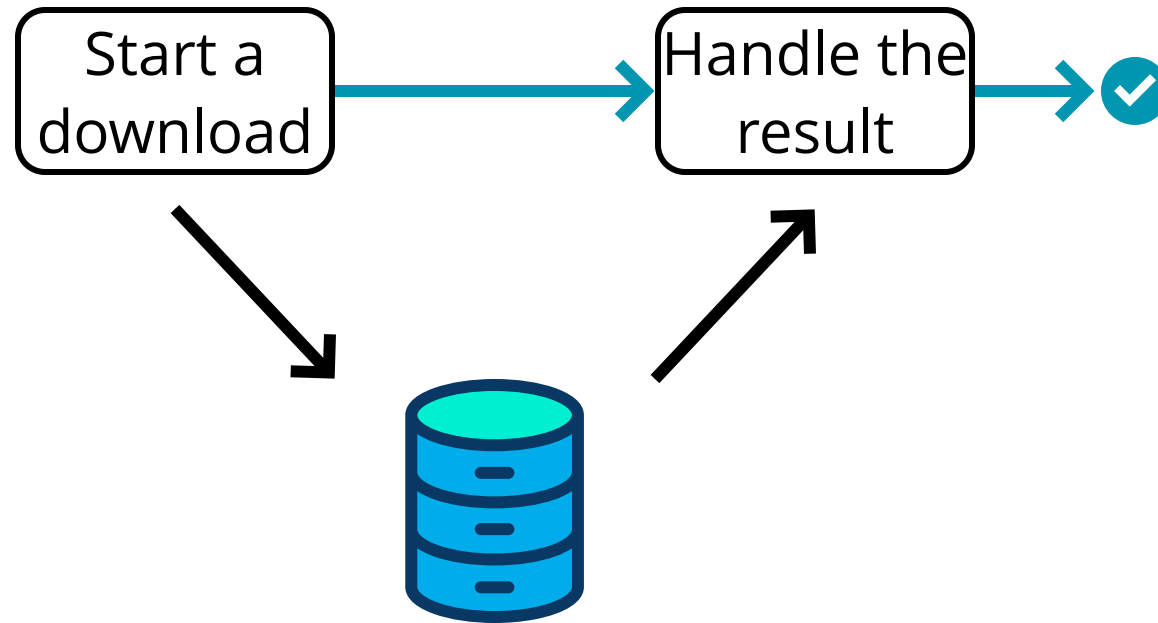
Our progress



Our progress

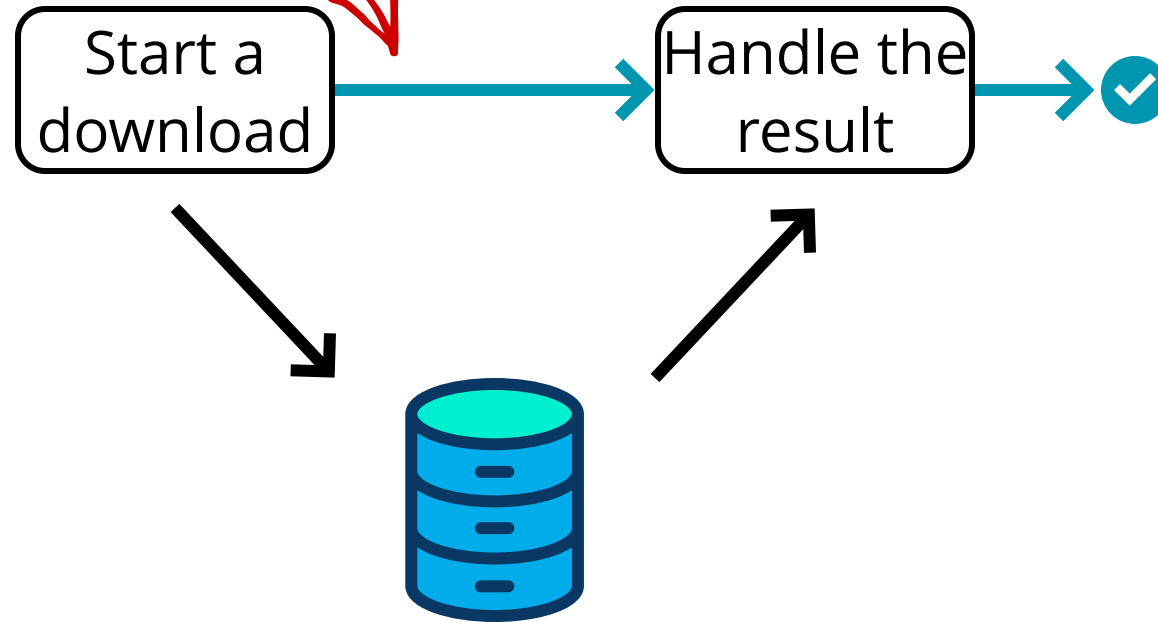


Coroutines

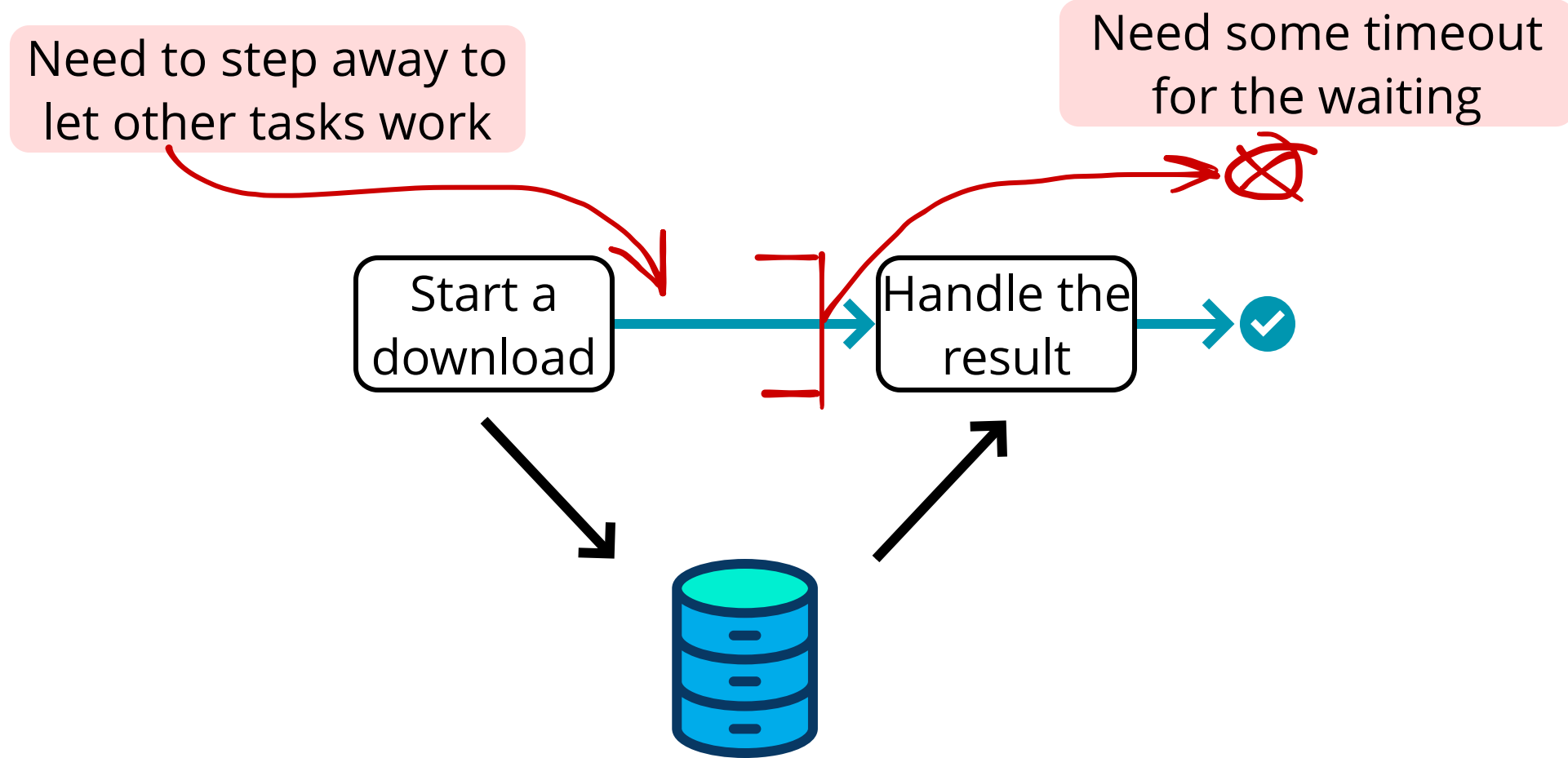


Coroutines

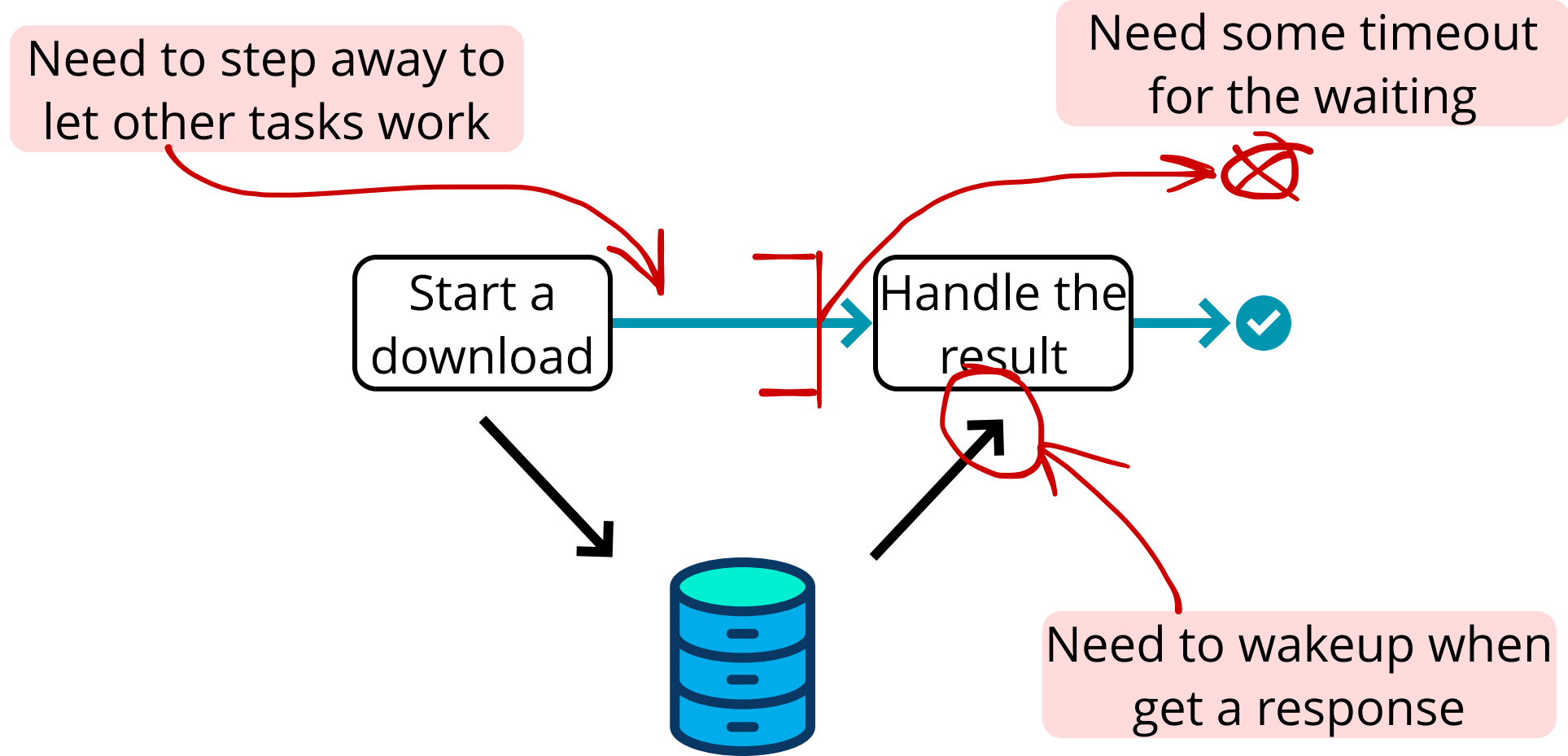
Need to step away to
let other tasks work



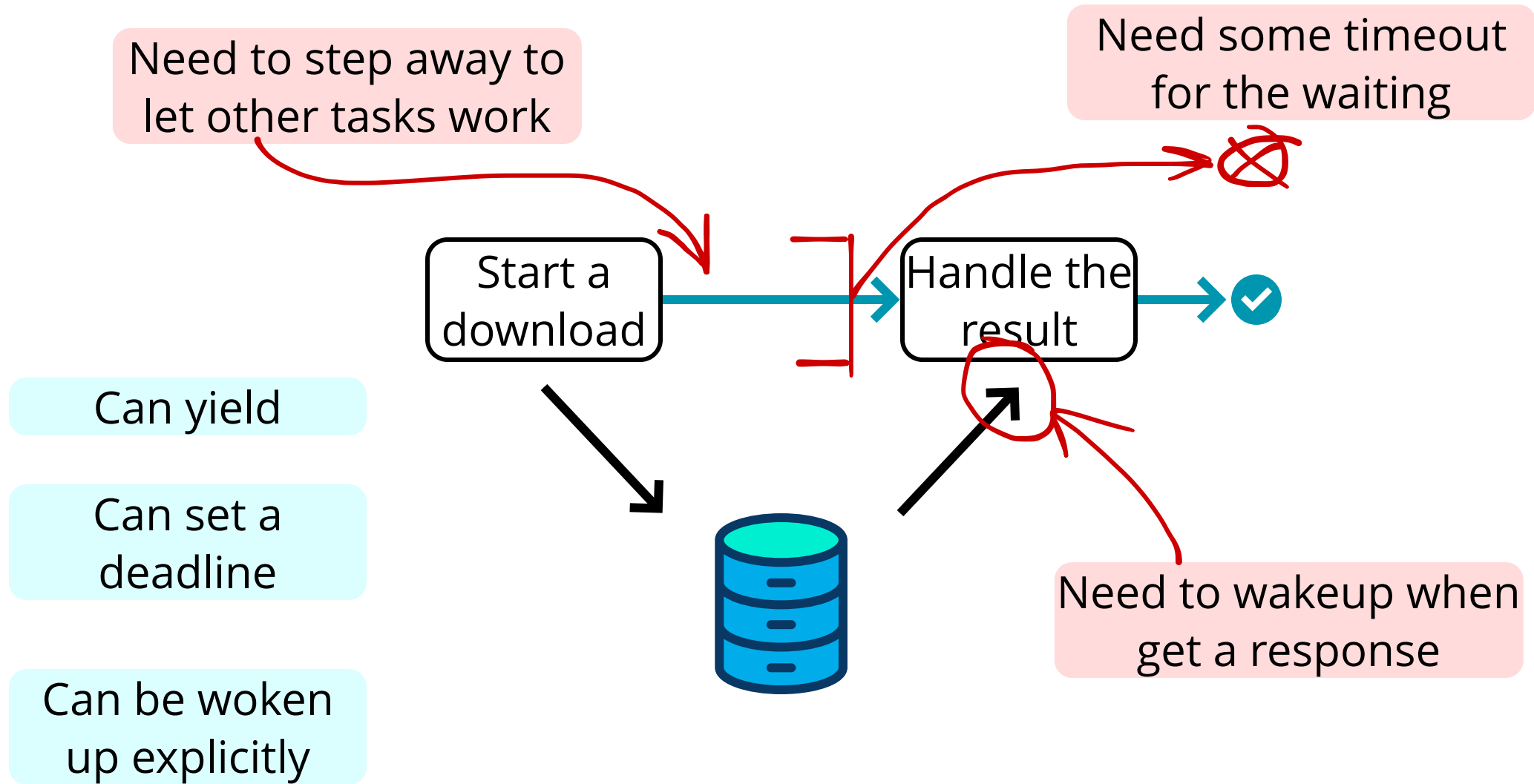
Coroutines



Coroutines



Coroutines



Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

Prepare the next
step

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

Send an async
request,
wakeup() on
completion

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

Yield until +5
secs or wakeup



Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (t->IsExpired()) {  
        http->Cancel();  
        sched.PostWait(t);  
        return;  
    }  
    if (http->IsSuccess())  
        HandleSuccess();  
    else  
        HandleFailure();  
    delete t;
```

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (t->IsExpired()) {  
        http->Cancel();  
        sched.PostWait(t);  
        return;  
    }  
    if (http->IsSuccess())  
        HandleSuccess();  
    else  
        HandleFailure();  
    delete t;
```

Check if the
deadline is up.
Cancel then

Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (t->IsExpired()) {  
        http->Cancel();  
        sched.PostWait(t);  
        return;  
    }  
    if (http->IsSuccess())  
        HandleSuccess();  
    else  
        HandleFailure();  
    delete t;
```

Not expired =
completed.
Handle it



Coroutines

```
TaskScheduler sched;  
HttpClient http;
```

```
MyTask *t = new MyTask();  
t->SetCallback(Download);  
sched.Post(t);
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url, {  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (t->IsExpired()) {  
        http->Cancel();  
        sched.PostWait(t);  
        return;  
    }  
    if (http->IsSuccess())  
        HandleSuccess();  
    else  
        HandleFailure();  
    delete t;
```

Completely lock-free,
very low overhead

**How to verify a
multithreaded
algorithm?**

TLA+ Verification

Temporal Logic of Actions

Language

Math logic with "time"
concept

Runtime

TLA+ Verification

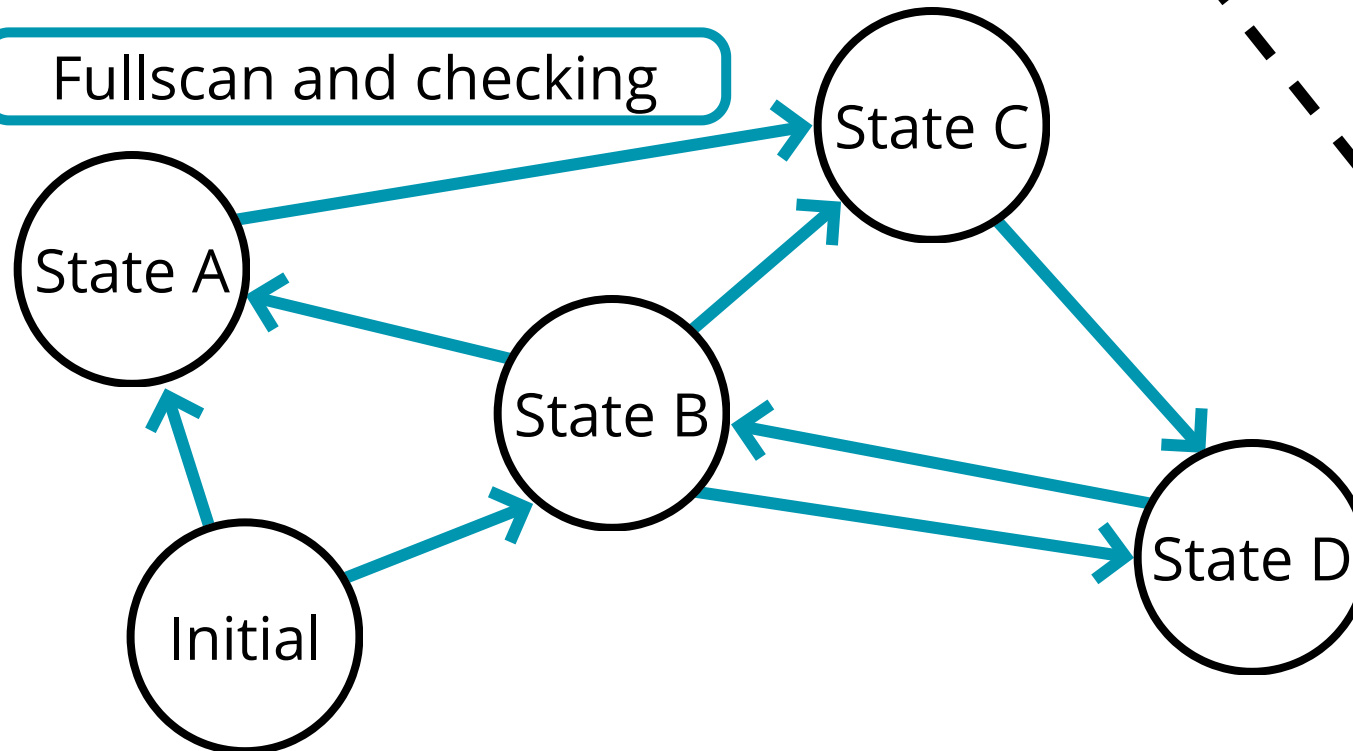
Temporal Logic of Actions

Language

Math logic with "time"
concept

Runtime

Fullscan and checking



Your system / algorithm

TLA+ Example

```
CONSTANT Count  
CONSTANT Lim  
VARIABLES Pipe,  
           LastReceived,  
           LastSent
```

Decide on
granularity of
the system:
objects, actions
on them

TLA+ Example

CONSTANT **Count**

CONSTANT **Lim**

VARIABLES **Pipe,**
 LastReceived,
 LastSent

```
Init ==  
  /\ Pipe = << >>  
  /\ LastReceived = 0  
  /\ LastSent = 0
```

Actions consist of
conditions. First
is usually "init"

TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
           LastReceived,
           LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0
```

Actions consist of
conditions. First
is usually "init"

Examples of
expressions:

X and Y are true:

$x \wedge y$

$X \wedge Y$

List of items x, y, z:

$\langle\langle x, y, z \rangle\rangle$

(x, y, z)

X equals Y:

$x = y$

$X = Y$

All items in set Y are equal 10:

$\forall x \in Y : x = 10$

$\forall x \in Y : x = 10$

TLA+ Example

CONSTANT **Count**

CONSTANT **Lim**

VARIABLES **Pipe,**
 LastReceived,
 LastSent

Init ==

 /\ **Pipe** = << >>
 /\ **LastReceived** = 0
 /\ **LastSent** = 0

Send ==

 /\ **Len(Pipe)** < **Lim**
 /\ **LastSent** < **Count**
 /\ **Pipe'** = **Append(Pipe, LastSent + 1)**
 /\ **LastSent'** = **LastSent + 1**

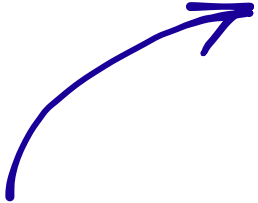
TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
          LastReceived,
          LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1
```

Conditions for
the action to be
possible



TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
           LastReceived,
           LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1
```

Conditions which
"change" the
state if the action
is possible

TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
           LastReceived,
           LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1
```

Single quote `'`
refers to the next
value of the
variable

Next value of `X` equals `X + 1`:
 $X' = X + 1$

TLA+ Example

CONSTANT **Count**

CONSTANT **Lim**

VARIABLES **Pipe,**
 LastReceived,
 LastSent

Init ==

 /\ **Pipe** = << >>
 /\ **LastReceived** = 0
 /\ **LastSent** = 0

Send ==

 /\ **Len(Pipe)** < **Lim**
 /\ **LastSent** < **Count**
 /\ **Pipe'** = **Append(Pipe, LastSent + 1)**
 /\ **LastSent'** = **LastSent + 1**

Recv ==

 /\ **Len(Pipe)** > 0
 /\ **LastReceived'** = **Head(Pipe)**
 /\ **Pipe'** = **Tail(Pipe)**

TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
          LastReceived,
          LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1

Recv ==
  /\ Len(Pipe) > 0
  /\ LastReceived' = Head(Pipe)
  /\ Pipe' = Tail(Pipe)

PipeInvariant ==
  /\ \A i \in 1..Len(Pipe) - 1: Pipe[i] + 1 = Pipe[i + 1]
  /\ Len(Pipe) =< Lim
  /\ \/ Len(Pipe) = 0
     \/ Pipe[1] = LastReceived + 1
```

TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
          LastReceived,
          LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1
```

```
Recv ==
  /\ Len(Pipe) > 0
  /\ LastReceived' = Head(Pipe)
  /\ Pipe' = Tail(Pipe)
```

Items are
ordered

```
PipeInvariant ==
  /\ \A i \in 1..Len(Pipe) - 1: Pipe[i] + 1 = Pipe[i + 1]
  /\ Len(Pipe) =< Lim
  /\ \/ Len(Pipe) = 0
  \/ Pipe[1] = LastReceived + 1
```

TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
          LastReceived,
          LastSent
```

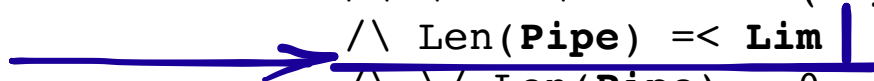
```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1

Recv ==
  /\ Len(Pipe) > 0
  /\ LastReceived' = Head(Pipe)
  /\ Pipe' = Tail(Pipe)

PipeInvariant ==
  /\ \A i \in 1..Len(Pipe) - 1: Pipe[i] + 1 = Pipe[i + 1]
  /\ Len(Pipe) =< Lim
  /\ \/ Len(Pipe) = 0
  /\ Pipe[1] = LastReceived + 1
```

The queue never
overflows



TLA+ Example

```
CONSTANT Count
CONSTANT Lim
VARIABLES Pipe,
          LastReceived,
          LastSent
```

```
Init ==
  /\ Pipe = << >>
  /\ LastReceived = 0
  /\ LastSent = 0

Send ==
  /\ Len(Pipe) < Lim
  /\ LastSent < Count
  /\ Pipe' = Append(Pipe, LastSent + 1)
  /\ LastSent' = LastSent + 1

Recv ==
  /\ Len(Pipe) > 0
  /\ LastReceived' = Head(Pipe)
  /\ Pipe' = Tail(Pipe)
```

```
PipeInvariant ==
  /\ \A i \in 1..Len(Pipe) - 1: Pipe[i] + 1 = Pipe[i + 1]
  /\ Len(Pipe) =< Lim
  /\ \ / Len(Pipe) = 0
  /\ Pipe[1] = LastReceived + 1
```

The first item is
always the next
to receive (**or** the
queue is empty)



TaskScheduler TLA+

MCSP queue spec: ~430 lines

[./tla/MCSPQueue.tla](#)

TaskScheduler spec: ~750 lines

[./tla/TaskScheduler.tla](#)

How to **run** it:

[./tla/README.md](#)

Study TLA+, great course from its author:

lamport.azurewebsites.net/tla/tla.html

Benchmarks - parts

Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

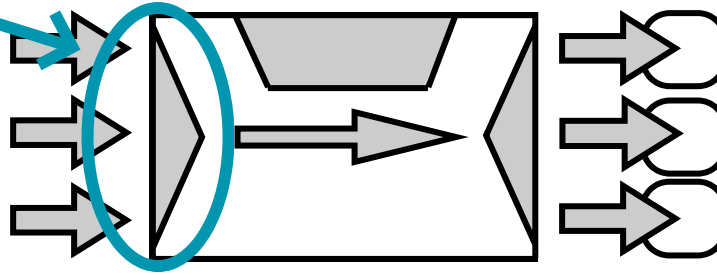
Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

Front Queue

- **5 push-threads**
- **~9 mln/sec**
- **x1.5** faster vs trivial



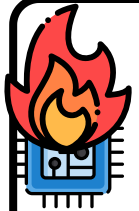
Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

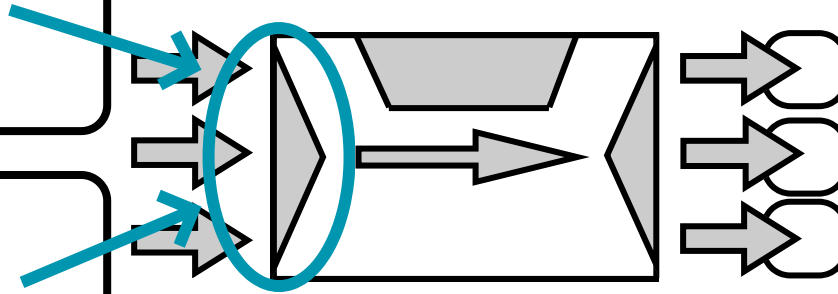
Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

Front Queue

- **5 push-threads**
- **~9 mln/sec**
- **x1.5** faster vs trivial



- **10 push-threads**
- **~5.8 mln/sec**
- **x2.6** faster vs trivial



Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

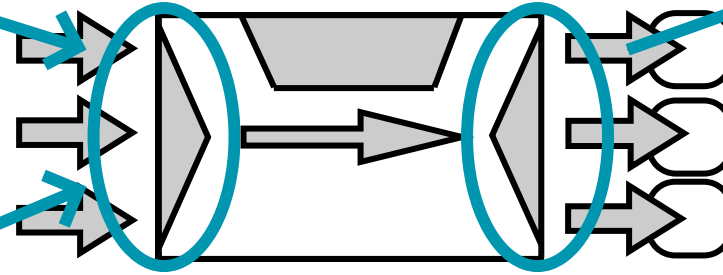
Front Queue

Ready Queue

- **5 push-threads**
- **~9 mln/sec**
- **x1.5** faster vs trivial



- **10 push-threads**
- **~5.8 mln/sec**
- **x2.6** faster vs trivial



- **5 pop-threads**
- **~2.5 mln/sec**
- **x2.6** faster vs trivial
- **x0.0009** lock-contention

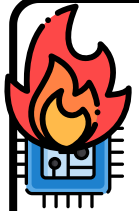
Benchmarks - parts

Comparative - **algorithm** vs its **naive trivial version**

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

Front Queue

- **5 push-threads**
- **~9 mln/sec**
- **x1.5** faster vs trivial



- **10 push-threads**
- **~5.8 mln/sec**
- **x2.6** faster vs trivial

Ready Queue

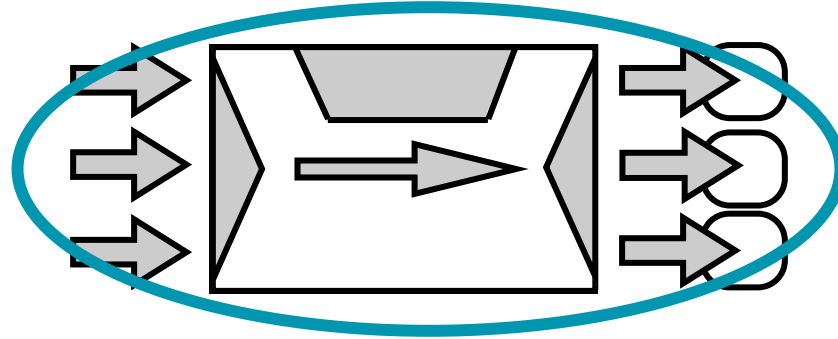
- **5 pop-threads**
- **~2.5 mln/sec**
- **x2.6** faster vs trivial
- **x0.0009** lock-contention

- **10 pop-threads**
- **~1.7 mln tasks/sec**
- **x4.5** faster vs trivial
- **x0.0007** lock-contention



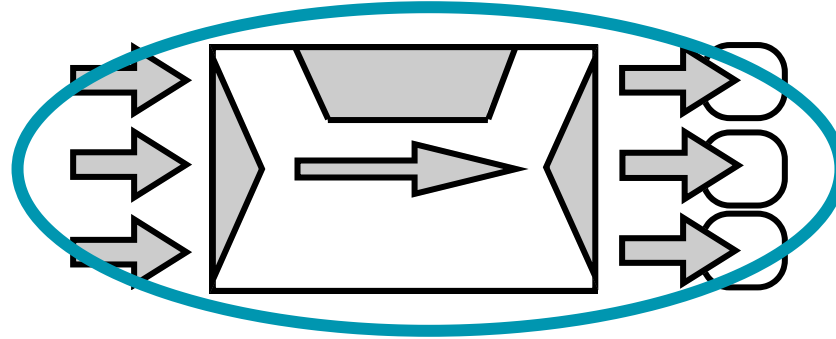
Benchmarks - scheduler

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading



Benchmarks - scheduler

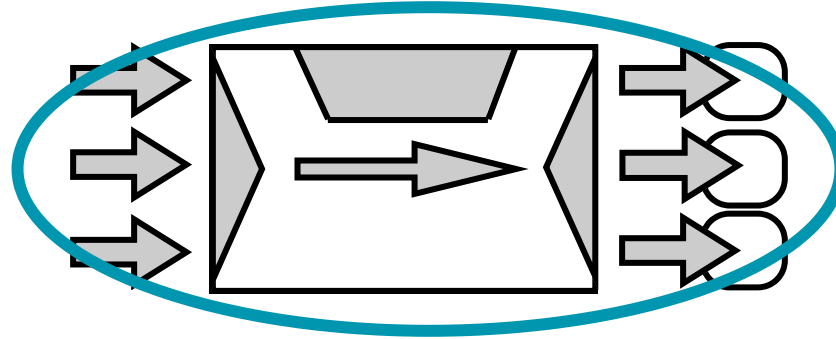
Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading



- 1 worker
- ~11 mln/sec
- **x2.2 faster** vs trivial
- **x0 lock**-contention

Benchmarks - scheduler

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading

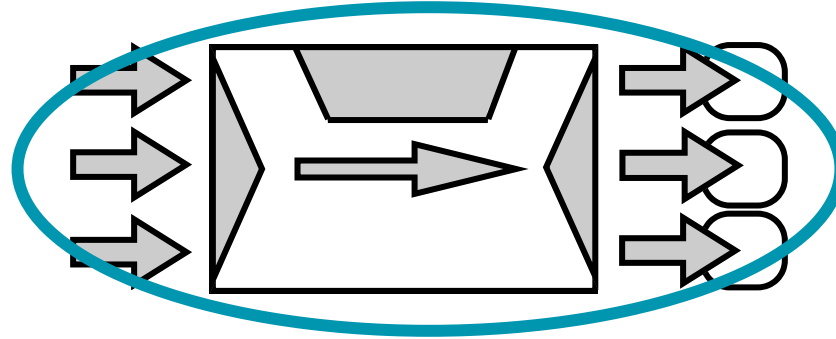


- 1 worker
- ~11 mln/sec
- **x2.2 faster** vs trivial
- **x0 lock**-contention

- 5 workers
- ~4 mln/sec
- **x3 faster** vs trivial
- **x0.002 lock**-contention

Benchmarks - scheduler

Example: Debian Linux, 8 cores, 2.3GHz, hyperthreading



- 1 worker
- ~11 mln/sec
- **x2.2 faster** vs trivial
- **x0 lock**-contention

- 5 workers
- ~4 mln/sec
- **x3 faster** vs trivial
- **x0.002 lock**-contention

- 10 workers
- ~5.2 mln/sec
- **x7.5 faster** vs trivial
- **x0.003 lock**-contention



Real usage

Savegame blobs multistep_processing

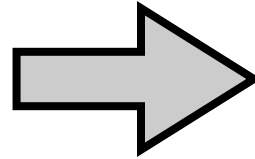
Real usage

Savegame blobs multistep processing

Debian Linux, 8 cores, 2.3GHz, hyperthreading

"Updater":

- 4 workers
- ~**100-300** RPS
- ~**500** ms latency



x10 speed
up right away

"TaskScheduler":

- 4 workers
- >**10000** RPS
- ~**100** ms latency

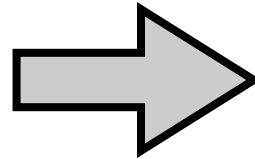
Real usage

Savegame blobs multistep processing

Debian Linux, 8 cores, 2.3GHz, hyperthreading

"Updater":

- 4 workers
- ~**100-300** RPS
- ~**500** ms latency

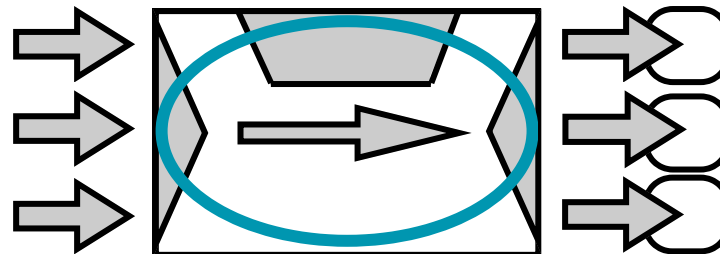


x10 speed
up right away

"TaskScheduler":

- 4 workers
- >**10000** RPS
- ~**100** ms latency

The algorithm is extendible



This is a **thread-safe box**. Can do here anything, not just deadlines. Like add **epoll** or **IOCP**

Future plans

Try on ARM

Other languages

Optimizations

The End

The End

TLA+ specs and C++ code:

[github.com/ubisoft/
task-scheduler](https://github.com/ubisoft/task-scheduler)

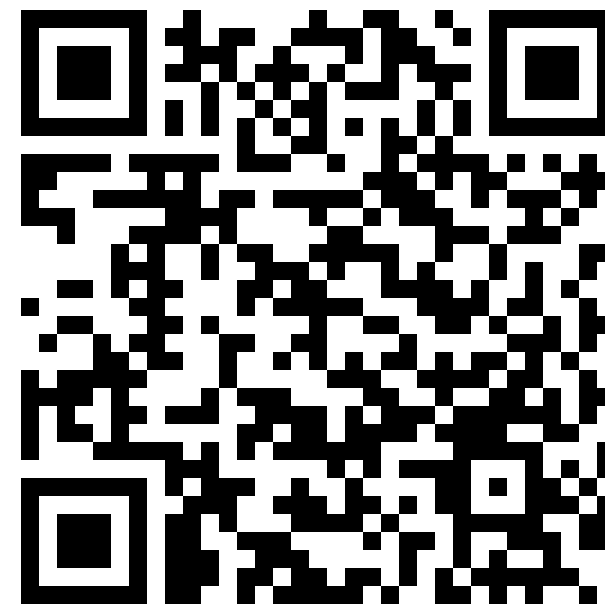
My talks (and this one too):

[slides.com/gerold103/
taskscheduler-highload2022-eng](https://slides.com/gerold103/taskscheduler-highload2022-eng)

How do the
workers sleep?

Smart usage of
cmpxchg()

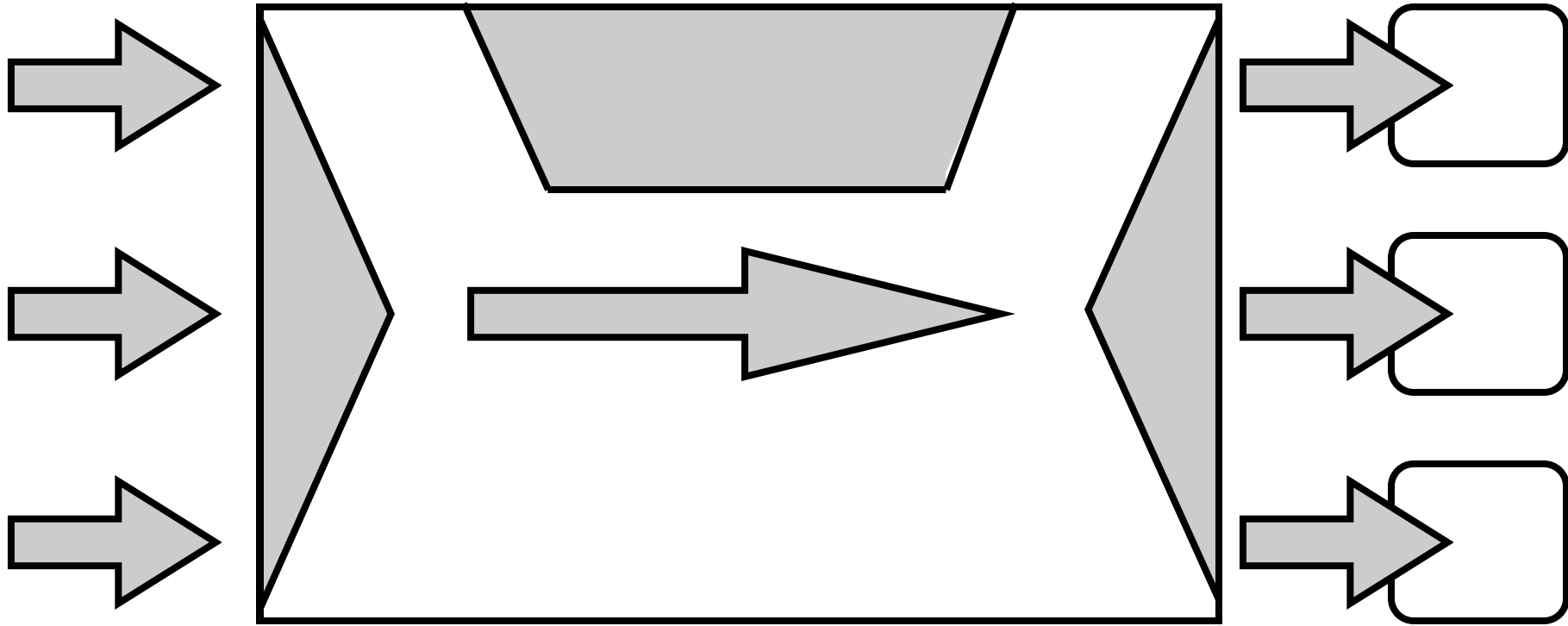
Coroutine
Wakeup vs Signal



Feedback

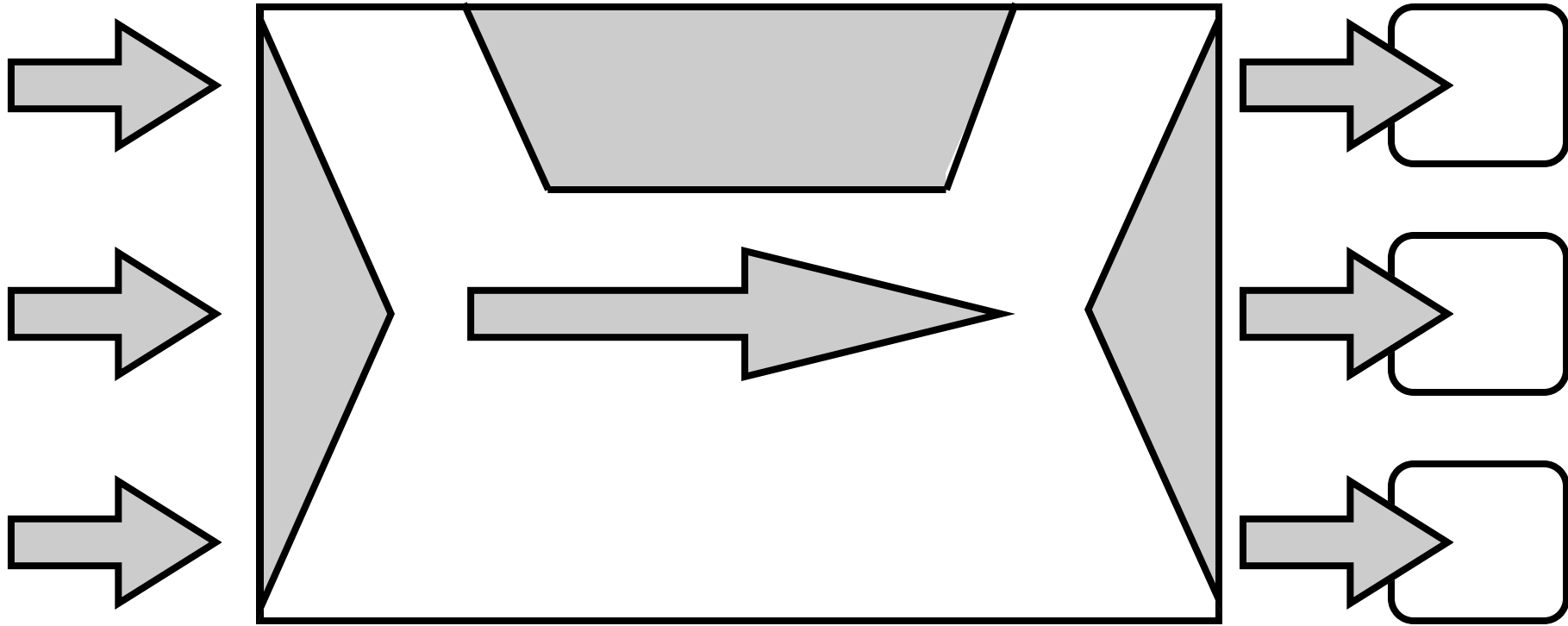
Additional content

Signal



Signal

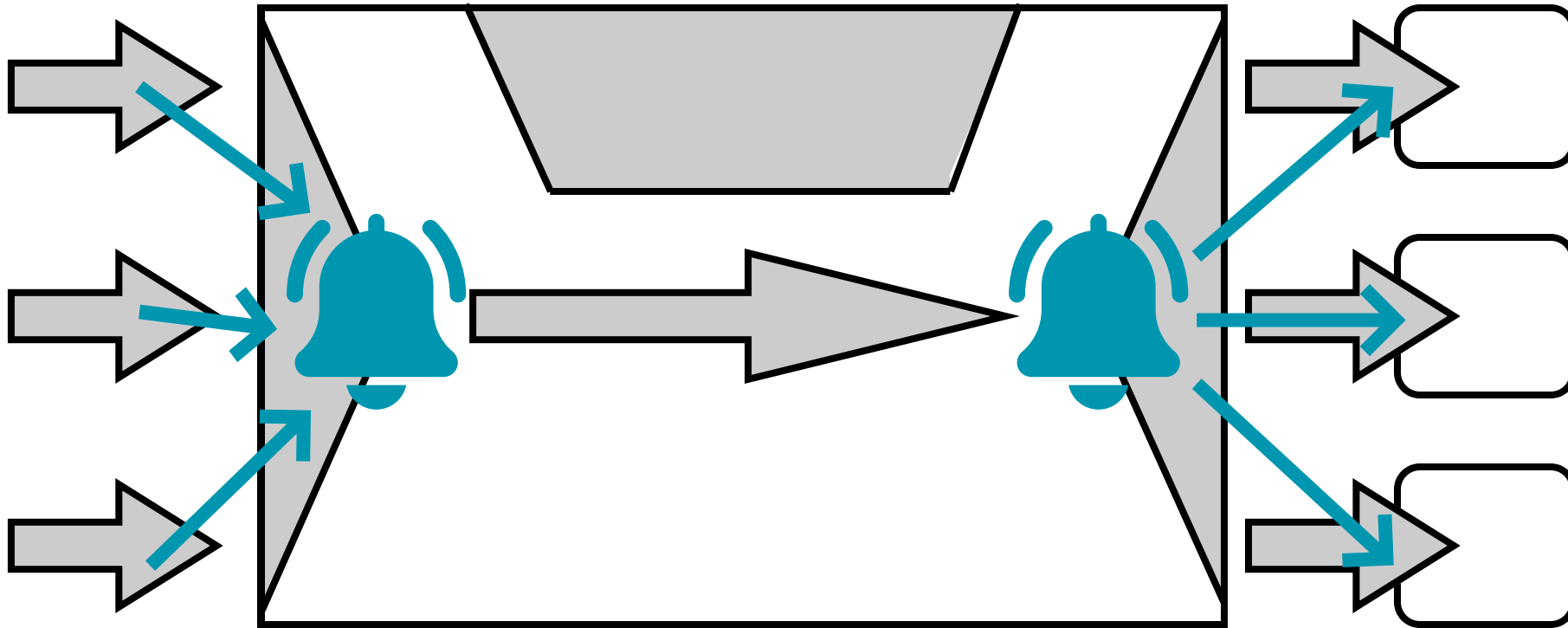
Threads shouldn't **poll**

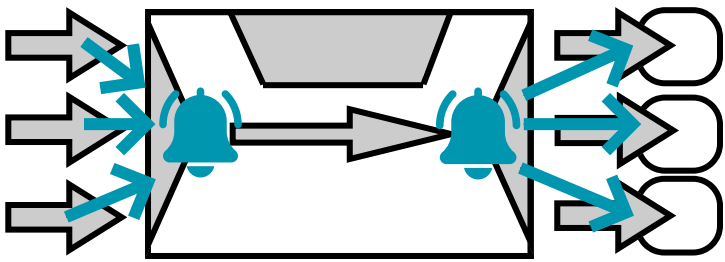


Signal

An event storage

Threads shouldn't **poll**





Signal

An event storage

Threads shouldn't **poll**

```
Signal sig;  
bool hasEvent = false;
```

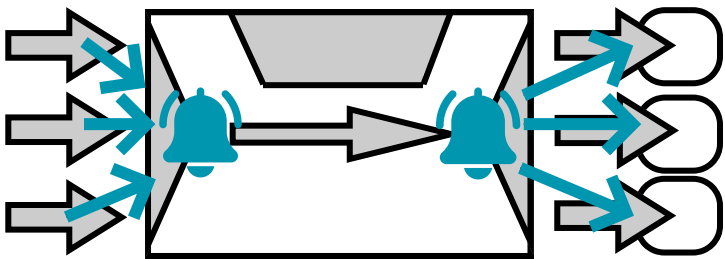
Thread 1:

```
hasEvent = true;  
sig.Send();
```

Thread 2:

```
sig.BlockingReceive();  
assert(hasEvent);  
// Won't receive again:  
assert(sig.IsEmpty(););
```





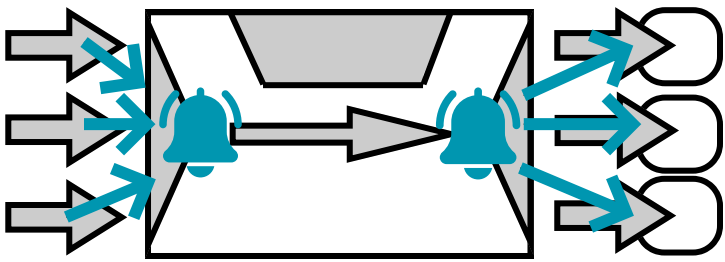
Signal

An event storage

Threads shouldn't **poll**

Usual implementation

```
class Signal:  
    Mutex myLock;  
    ConditionVariable myCond;  
    bool myFlag;
```



Signal

An event storage

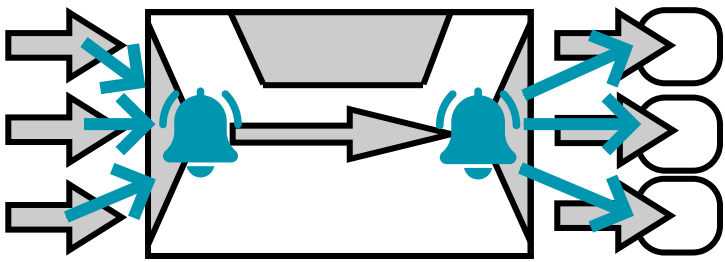
Threads shouldn't **poll**

Usual implementation

```
class Signal:  
    Mutex myLock;  
    ConditionVariable myCond;  
    bool myFlag;
```

```
Signal::Send()  
{  
    myLock.Lock();  
    myFlag = true;  
    myCond.Signal();  
    myLock.Unlock();  
}
```

```
Signal::BlockingReceive()  
{  
    myLock.Lock();  
    while (not myFlag)  
        myCond.Wait();  
    myFlag = false;  
    myLock.Unlock();  
}
```



Signal

An event storage

Threads shouldn't **poll**

Usual implementation

```
class Signal:  
    Mutex myLock;  
    ConditionVariable myCond;  
    bool myFlag;
```

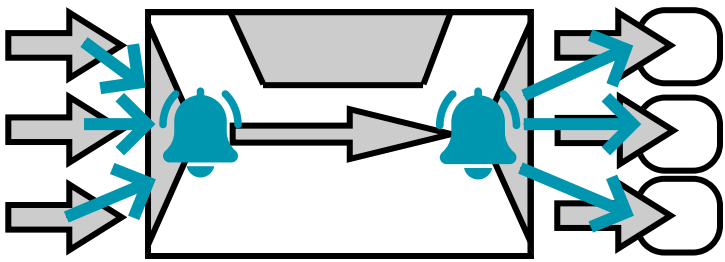
```
Signal::Send()
```

```
{  
    myLock.Lock();  
    myFlag = true;  
    myCond.Signal();  
    myLock.Unlock();  
}
```

Expensive
mutex lock
on each
operation

```
Signal::BlockingReceive()
```

```
{  
    myLock.Lock();  
    while (not myFlag)  
        myCond.Wait();  
    myFlag = false;  
    myLock.Unlock();  
}
```



Signal

An event storage

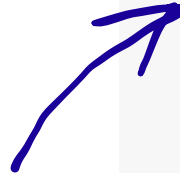
Threads shouldn't **poll**

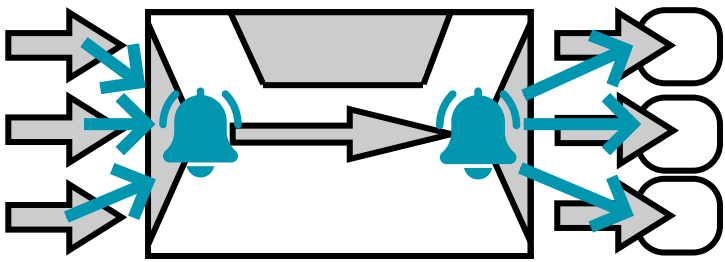
Lock-free receipt if
already signaled

```
class Signal:  
    Mutex myLock;  
    ConditionVariable myCond;  
    bool myFlag;
```

```
Signal::Send()  
{  
    myLock.Lock();  
    AtomicExchange(myFlag, true);  
    myCond.Signal();  
    myLock.Unlock();  
}
```

```
Signal::BlockingReceive()  
{  
    if (AtomicExchange(myFlag, false))  
        return;  
    myLock.Lock();  
    while (not AtomicExchange(myFlag, false))  
        myCond.Wait();  
    myLock.Unlock();  
}
```





Signal

An event storage

Threads shouldn't **poll**

Lock-free receipt if
already signaled

Lock-free send if
already signaled

```
class Signal:  
    Mutex myLock;  
    ConditionVariable myCond;  
    bool myFlag;
```

Signal::Send()

```
{  
    if (AtomicExchange(myFlag, true))  
        return;  
    myLock.Lock();  
    myCond.Signal();  
    myLock.Unlock();  
}
```

Signal::BlockingReceive()

```
{  
    if (AtomicExchange(myFlag, false))  
        return;  
    myLock.Lock();  
    while (not AtomicExchange(myFlag, false))  
        myCond.Wait();  
    myLock.Unlock();  
}
```

Back

Smart usage of cmpxchg

```
AtomicCompareExchange(var, new_value, check)
{
    if (old_value == check)
        return false;
    var = new_value;
    return true;
}
```

Smart usage of cmpxchg

```
AtomicCompareExchangeGetOld(var, new_value, check)
{
    if (old_value == check)
        return old_value;
    var = new_value;
    return old_value;
}
```

Smart usage of cmpxchg

```
AtomicCompareExchangeGetOld(var, new_value, check)
{
    if (old_value == check)
        return old_value;
    var = new_value;
    return old_value;
}
```

```
AtomicCompareExchange(var, new_value, check)
{
    return AtomicCompareExchangeGetOld(
        val, new_value, check) == check;
}
```

Smart usage of cmpxchg

```
class MPSCQueue:  
    T* myTop;
```

```
MPSCQueue::Push(T* aItem)  
{  
    T* oldTop;  
    do {  
        oldTop = AtomicLoad(myTop);  
        aItem->myNext = oldTop;  
    } while (not AtomicCompareExchange(  
        myTop, aItem, oldTop));  
}  
  
MPSCQueue::PopAll(T* aItem)  
{  
    T* top = AtomicExchange(myTop, nullptr);  
    return ReverseList(top);  
}
```

Smart usage of cmpxchg

```
class MPSCQueue:  
    T* myTop;
```

```
MPSCQueue::Push(T* aItem)  
{  
    T* oldTop;  
    do {  
        oldTop = AtomicLoad(myTop);  
        aItem->myNext = oldTop;  
    } while (not AtomicCompareExchange(  
        myTop, aItem, oldTop));  
}  
  
MPSCQueue::PopAll(T* aItem)  
{  
    T* top = AtomicExchange(myTop, nullptr);  
    return ReverseList(top);  
}
```

Smart usage of cmpxchg

```
class MPSCQueue:  
    T* myTop;
```

Load the top
once

```
MPSCQueue::Push(T* aItem)  
{
```

```
    T* oldTop;
```

```
    T* res = AtomicLoad(myTop);
```

```
    do {
```

```
        oldTop = res;
```

```
        aItem->myNext = oldTop;
```

```
        res = AtomicCompareExchangeGetOld(myTop, aItem, oldTop);
```

```
    } while (res != oldTop);
```

```
MPSCQueue::PopAll(T* aItem)  
{
```

```
    T* top = AtomicExchange(myTop, nullptr);
```

```
    return ReverseList(top);  
}
```

Atomically retry
setting a new top and
getting the old one

Back



Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
        http->SetReady();  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (http->IsReady())  
    {  
        Process(http->GetResult());  
        delete t;  
        return;  
    }  
    http->Cancel();  
    sched.PostWait(t);
```

Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
        http->SetReady();  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (http->IsReady())  
    {  
        Process(http->GetResult());  
        delete t;  
        return;  
    }  
    http->Cancel();  
    sched.PostWait(t);
```

Check the
completion
before the
expiration

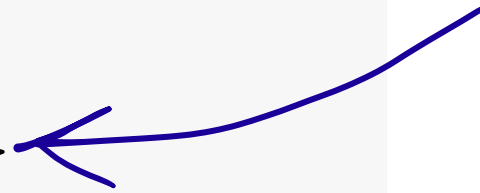
Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
        http->SetReady();  
        sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (http->IsReady())  
    {  
        Process(http->GetResult());  
        delete t;  
        return;  
    }  
    http->Cancel();  
    sched.PostWait(t);
```

The code will
eventually crash
here

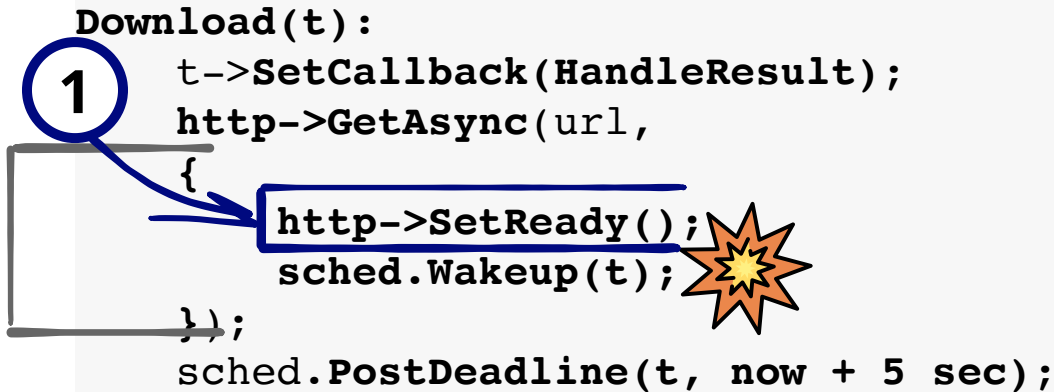


Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

Thread A

```
Download(t):  
  1 t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
      http->SetReady();  
      sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```



HTTP threads set
'ready' flag

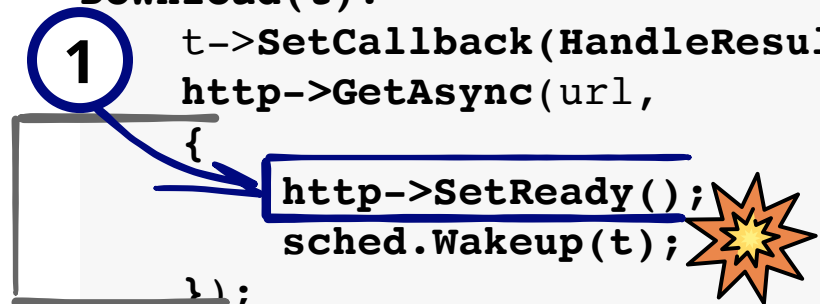
```
HandleResult(t):  
  if (http->IsReady())  
  {  
    Process(http->GetResult());  
    delete t;  
    return;  
  }  
  http->Cancel();  
  sched.PostWait(t);
```

Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

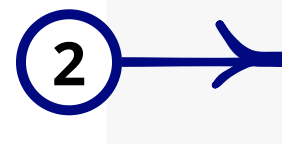
Thread A

```
Download(t):  
  ① t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
      http->SetReady();  
      sched.Wakeup(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```



Thread B

```
HandleResult(t):  
  if (http->IsReady())  
  {  
    ② Process(http->GetResult());  
    delete t;  
    return;  
  }  
  http->Cancel();  
  sched.PostWait(t);
```



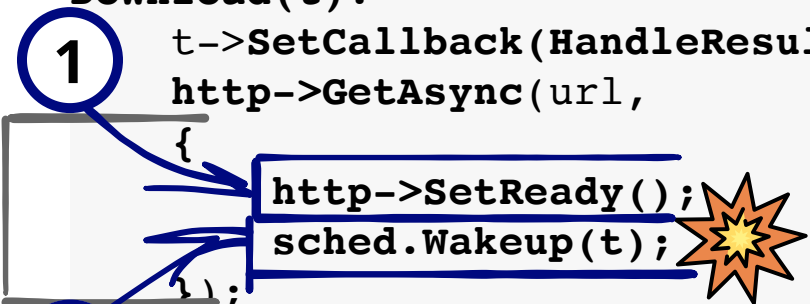
Scheduler thread
wakes up by
timeout and
deletes the task

Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

Thread A

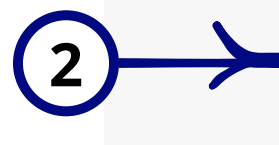
```
Download(t):  
  ① t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
      http->SetReady();  
      sched.Wakeup(t);  
    },  
  ③ sched.PostDeadline(t, now + 5 sec);
```



HTTP thread uses a
deleted task

Thread B

```
HandleResult(t):  
  if (http->IsReady())  
  {  
    ② Process(http->GetResult());  
    delete t;  
    return;  
  }  
  http->Cancel();  
  sched.PostWait(t);
```



Task signal

```
TaskScheduler sched;  
HttpClient http;
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
        sched.Signal(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

```
HandleResult(t):  
    if (t->ReceiveSignal())  
    {  
        Process(http->GetResult());  
        delete t;  
        return;  
    }  
    http->Cancel();  
    sched.PostWait(t);
```



Task signal

```
TaskScheduler sched;  
HTTPClient http;
```

```
Download(t):  
    t->SetCallback(HandleResult);  
    http->GetAsync(url,  
    {  
        sched.Signal(t);  
    });  
    sched.PostDeadline(t, now + 5 sec);
```

Signal is atomic
"wakeup + set flag"

```
HandleResult(t):  
    if (t->ReceiveSignal())  
    {  
        Process(http->GetResult());  
        delete t;  
        return;  
    }  
    http->Cancel();  
    sched.PostWait(t);
```

Back